PARALLEL ALGORITHMS FOR GENERALIZED N-BODY PROBLEM IN HIGH DIMENSIONS AND THEIR APPLICATIONS FOR BAYESIAN INFERENCE AND IMAGE ANALYSIS

A Thesis
Presented to
The Academic Faculty

by

Bo Xiao

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology
December 2014

PARALLEL ALGORITHMS FOR GENERALIZED N-BODY PROBLEM IN HIGH DIMENSIONS AND THEIR APPLICATIONS FOR BAYESIAN INFERENCE AND IMAGE ANALYSIS

Approved by:

Professor Edmond Chow,
Committee Chair
School of Computational Science and Engineering
Georgia Institute of Technology

Professor George Biros, Advisor
Institute for Computational Engineering and Sciences
The University of Texas at Austin

Professor Hongyuan Zha
School of Computational Science and Engineering
Georgia Institute of Technology

Professor David Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Bistra Dilkina
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: 18th August 2014

To my parents,my wife and my son.

ACKNOWLEDGEMENTS

The efforts and support of many individuals have made it a pleasant and unforgettable journey. I owe an enormous debt of gratitude to my advisor Dr. George Biros, who have showed me the beauty of science, offered me precious opportunities for collaborations, and supplied sustenance, literally and metaphorically. His insight, enthusiasm and leadership are always the inspiration for me. I am grateful to Dr. Hongyuan Zha for numerous insightful discussions during the process of my research. I would like to thank my thesis committee members, Dr. Edmond Chow, Dr. David A. Bader and Dr. Bistra Dilkina, for their encouragement and suggestions, as well as the friendly accessibility throughout my graduate study.

I am thankful to the members in the PADAS group: Amir Gholaminejad, Dhairya Malhotra, Bill March, Mang Andreas, Susan M. Rodriguez and Taylor Donna. You provide a great environment for research and friendship. It is my fortunate to know all of you.

I am deeply indebted to my parents for bringing me into this world and their unqualified love and support during my life. I would like to thank my wife and my baby boy, to make my PhD life much colorful and enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this dissertation, we explore parallel algorithms for general N-Body problems in high dimensions, and their applications in machine learning and image analysis on distributed infrastructures.

In the first part of this work, we proposed and developed a set of basic tools built on top of Message Passing Interface and OpenMP for massively parallel nearest neighbors search. In particular, we present a distributed tree structure to index data in arbitrary number of dimensions, and a novel algorithm that eliminate the need for collective coordinate exchanges during tree construction. To the best of our knowledge, our nearest neighbors package is the first attempt that scales to millions of cores in up to a thousand dimensions.

Based on our nearest neighbors search algorithms, we present "ASKIT", a parallel fast kernel summation tree code with a new near-far field decomposition and a new compact representation for the far field. Specially our algorithm is kernel independent. The efficiency of new near far decomposition depends only on the intrinsic dimensionality of data, and the new far field representation only relies on the rand of sub-blocks of the kernel matrix.

In the second part, we developed a Bayesian inference framework and a variational formulation for a MAP estimation of the label field for medical image segmentation. In particular, we propose new representations for both likelihood probability and prior probability functions, as well as their fast calculation. Then a parallel matrix free optimization algorithm is given to solve the MAP estimation. Our new prior function is suitable for lots of spatial inverse problems. Experimental results show our framework is robust to noise, variations of shapes and artifacts.

CHAPTER I

INTRODUCTION

In this dissertation, we explore some basic computational problems in machine learning on distributed infrastructures. Data are usually distributed since the number of points is too large to fit into one single CPU and parallelism can allow high scalability for distributed queries. Each process only owns a subset of the whole data, during the computations every process has to communicate with other processes to share its information. Lots of machine learning techniques and scientific simulations require kind of parallelism and distribution of data to complete their work in a reasonable time. Unlike the two or three dimensional setting in the usual N-body problem, the dimensionality of data can be as high as from several hundreds to even more than several thousands for many real machine learning applications. My dissertation attempts to provide some scalable building blocks to accelerate the computations with a large scale of data in high dimensions.

1.1 Motivation

The work in this dissertation is motivated by the investigation of medical image segmentation. Image segmentation is the process of labeling each pixel in an image based on the anatomical structure to which it corresponds. Unlike the general segmentation problems in image processing such as landscapes or surveillance images, medical image segmentation has its own characteristics. First of all, medical image analysis requires high accuracy to help researcher and surgeons make decisions. The high precision requirement makes lots of clustering and thresholding approaches in general segmentation fail for medical images. Besides, due to the noise, artifacts, the similar appearance of different tissues in many modalities, and the complexity of anatomical structures, precise segmentation becomes an even tougher challenge. Studies of the consistency of segmentations across experts and the

reproducibility of the a segmentation by the same expert over time shows a high degree (15%) of variance [71]. On the other hand, for a specific tissue in a medical image, usually we could obtain some previous scans either of the same person or of other patients. Furthermore, we could have relevant anatomical information about this tissue. Those information forms lots of good prior knowledge. Fully making use of the existing knowledge about the target segmentation leads us to a supervise learning scheme.

In this work, we consider only the binary case, i.e., we try to segment the target object from background in an image. Given an image $\mathbf{I} \in \mathbb{R}^N$, where N is the number of pixels, we seek to find a label field $\boldsymbol{\phi} \in \{0,1\}^N$, where 0 indicates the background pixels and 1 indicates the object pixels. Let $\mathbf{I}(x)$ and $\boldsymbol{\phi}(x)$ indicate the discrete grayscale image and label value at the pixel x. A segmentation problem is then defined as: given the observed image $\mathbf{I}$, infer the label $\boldsymbol{\phi}$ for each pixel in $\mathbf{I}$.

Inferring $\boldsymbol{\phi}$ is always challenging due to the complexity both of the human anatomy and of the imaging techniques. Noise, artifacts, variability of patients all limit the single image processing routines give a satisfactory analysis. We adopt a statistical Bayesian variational framework to incorporate prior knowledge derived from a **training set**. Given a set of images $\{\mathbf{I}_i, \boldsymbol{\phi}_i\}, i = 1, \cdots, n$, where for each $\mathbf{I}_i$, we have already obtained its corresponding label $\boldsymbol{\phi}_i$ by expert manually labeling, we infer the label $\boldsymbol{\phi}$ from the posterior distribution $p(\boldsymbol{\phi}|\mathbf{I}) \sim p(\mathbf{I}|\boldsymbol{\phi})p(\boldsymbol{\phi})$. The Maximum a posteriori (MAP) estimate is found by maximizing the log-posteriori:

$$\boldsymbol{\phi}^* = \text{argmax}_{\boldsymbol{\phi}} \log p(\mathbf{I}|\boldsymbol{\phi}) + \log p(\boldsymbol{\phi}) \tag{1}$$

where $p(\mathbf{I}|\boldsymbol{\phi})$ is the likelihood probability function and $p(\boldsymbol{\phi})$ is the prior probability function.

There are two main issues that one has to address for developing scalable algorithms to solve the MAP problem. First, models of probability $p(\mathbf{I}|\boldsymbol{\phi})$ and $p(\boldsymbol{\phi})$ can be quite rich. To be specific to the problem being solved and to live on less assumptions about the real distribution of those probabilities, we use a non parametric function based on kernel density

estimation (KDE). Secondly, achieving scalability in a distributed setting due to large scale of data and expensive computational cost requires additional considerations such as: 1) minimizing the critical time on each single section, 2) minimizing communication among processes. Those requirement encourages new distributed data structures and algorithms to accelerate the computations. In my dissertation, to evaluate KDE efficiently, it requires two building blocks: **nearest neighbors search** and **kernel summation**. We also need parallel optimization routines to solve Eqn. (1). The derived algorithms for nearest neighbors search and fast kernel summation routines are general and suitable for all other related applications in machine learning and scientific simulations. The optimization for the segmentation is easy to adapted to inverse problems for spacial field. For all these routines, I use OpenMP for shared memory (intra node) parallelism and message passing interface (MPI) for distributed memory (inter nodes) parallelism.

1.2 Problem Definitions

In summary, we try to solve these questions in this dissertation:

- k- and range nearest neighbors search (`KNN`): for each *query point* $\{q_j\}_{j=1}^m \in \mathbb{R}^d$, find the k-nearest neighbors or find all points $r \in \mathcal{R}$ such that $d(r, q) < \rho$, where ρ is the range, and $d(\cdot, \cdot)$ is the distance measure between two points. Figure 1 illustrates the nearest neighbors search problems. The 'star' refers to the query point q, the 4 nearest neighbors of q are marked as square. The points within the red circle are the ρ nearest neighbors of q in terms of range search.

- Fast kernel summation: Define weights $w_j \in \mathbb{R}$ and u_i, the accumulation sum of kernels, the kernel summation can be expressed as a form

$$u(\mathbf{x}) = \sum_{j=1}^{N} K(\mathbf{x}, \mathbf{x}_j) w_j, \quad \forall j = 1 \dots N. \tag{2}$$

  where w_j contains the bandwidth information.

- Modeling likelihood probability function $p(\mathbf{I}|\boldsymbol{\phi})$ and spatial prior probability function $p(\boldsymbol{\phi})$ based on the given training set.

- Medical image segmentation based on the training set: given a set of images with ground truth labels $\{\mathbf{I}_i, \boldsymbol{\phi}_i\}_{i=1}^n$, for each novel image $\mathbf{I}'$, we try to infer its corresponding label $\boldsymbol{\phi}'$ by finding the MAP solution of Eqn. (1). Figure 2 and Figure 3 gives examples of segmentation of a cardiac image and a brain image separately.

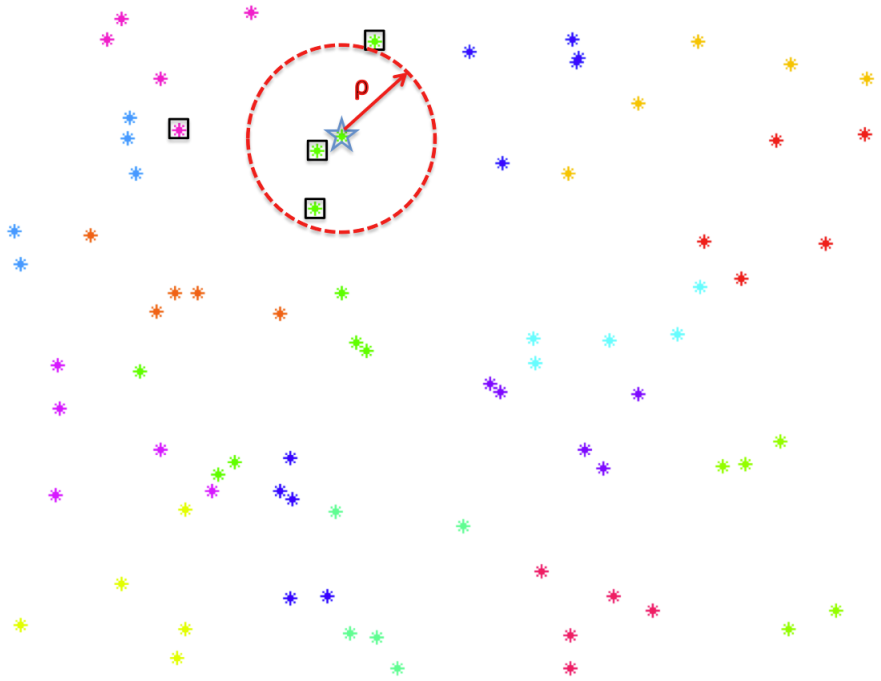
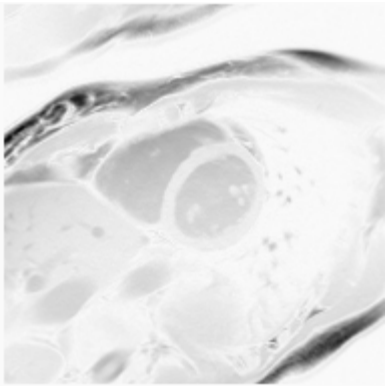


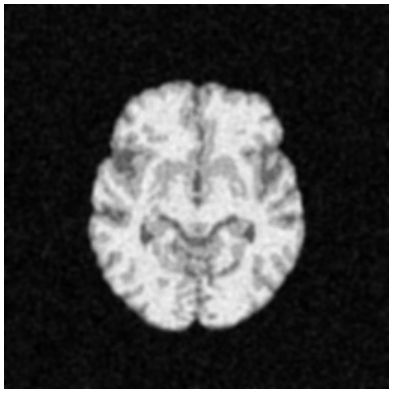
Figure 1: *Nearest neighbor problems:* In the nearest neighbor problem, we seek to identify the k nearest points (marked as 'square') that are closest to a query point $q \in \mathbb{R}^d$ (marked as 'star'), given a set of reference points $r \in \mathbb{R}^d$. Such searches also can be posed in terms of range, in which we seek to find all the points in a ball of radius ρ centered at the query point q (points inside the red circle). An exhaustive search algorithm scales linearly per query point. Tree algorithms scale logarithmically, but the performance (i.e., pruning) deteriorates quickly as the dimension d increases.



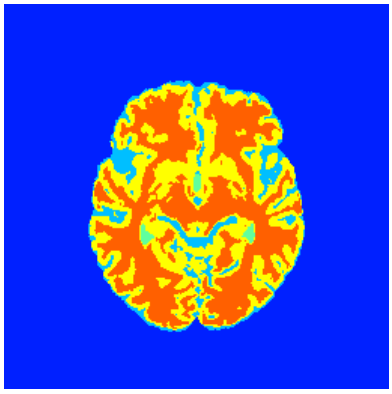
(a) cardiac image example

(b) segmentation of the cardiac image

Figure 2: *Examples of the cardiac image and its label.* To enhance the contrast of figure (a), the inverse image of the original one is illustrated. In figure (b), the color 'magenta' indicates the right ventricle and the color 'grey' indicates the left ventricle.



(a) brain image example



(b) segmentation of the brain image

Figure 3: *Examples of the brain image and its label.* In figure (b), the color 'yellow' indicates the grey matter and the color 'orange' indicates the white matter.

1.3 Challenges and Our Contributions

The author tough all the problems defined in §1.2 and had several contributions in each field in the form of published articles and software packages. The techniques and softwares are suitable for many applications in statistical learning, scientific simulation and image analysis. We use various numerical optimization methods and high performance computing techniques to develop the implementations. These algorithms are tested and tuned to scale to millions of cores for tera-bytes data in high dimensions (several hundreds). The components of this dissertation are summarized as the following:

Nearest neighbors search: The nearest neighbors problem has a trivial $\mathcal{O}(nm)$ algorithm, where m is the number of query points and n is the number of reference points. If $m = n$ (the problem is also known as the *all nearest neighbor* or *ANN*), direct search is prohibitively expensive. Spatial data structures can deliver $O(n \log n)$ or even $O(n)$ complexity for fixed dimension d. But for *high dimensions* (say, $d \geq 4$), spatial data structures provably deteriorate, and for very large d all known exact schemes end up having the complexity of the direct algorithm [129].

In many applications however, the "signal" is concentrated in a lower-dimensional manifold. In such cases, *approximate searches* using indexing techniques (tree-type data structures or hashing techniques) can perform better than direct searches. Although there is a lot of work on algorithms and implementations of such methods for distributed databases, there is limited work on algorithms and parallel scaling analysis on high-performance computing applications.

we study the performance of exact and approximate implementations as a function of the problem size, the number of dimensions. To the best of our knowledge, we made the first attempt to scale nearest neighbors search problem to millions of cores in up to 2000 dimensions. In particular,

- We propose two parallel algorithms for the direct calculation of the KNN problem. The first one trades memory in favor of computing time by replicating the query and reference points, minimizing the synchronization costs. For k-queries the work scales roughly as $\mathcal{O}(\frac{nm}{p} + k \log p)$ and its storage scales as $\mathcal{O}(mnp)$. The method is useful for small m or for the case in which wall-clock time is important. It is work-, but not memory-, optimal. The second algorithm uses a cyclic iteration whose time complexity scales as $\mathcal{O}(\frac{nm}{p} + m + n + \frac{mnk}{p})$ and the memory scales as $\mathcal{O}(nm)$.

- We present RKD TREE, a novel tree construction and traversal algorithm for indexing structures in arbitrary number of dimensions. Our method can be used for several

types of trees (e.g., ball trees, metric trees, spill trees, or KD-trees). It is a recursive, top-down algorithm in which every node corresponds to a group of processes. The key features of our method are that the tree is not replicated, it allows locality by minimizing the cost of collective communications, it is relatively easy to implement, and it can automatically switch to an approximate search when poor pruning performance is detected.

- We present a novel algorithm that eliminates the need for collective coordinate exchanges during construction which results in $2\times$–$4\times$ speedups on some machines and significantly increases the robustness of the algorithm.

- *Scalability tests on up to 240K cores*. We integrate single-core optimizations, shared memory parallelism (using OpenMP), distributed memory parallelism (using MPI), to enable calculations of unprecedented size, in terms of the dataset size (24 TB), number of dimensions (2,048), and number of cores (240K).

Table 1 gives a general view of our approaches. To be clear, it listed the largest problem size of each method we solved in our experiments. The million cycles per point per core are some typical values for a certain type of data points. Due to the internal structure and intrinsic dimensionality of data points, the efficiency and accuracy of metric tree and approximate approaches (LSH and RKD TREE) can vary differently, which we will discuss in more details in chapter 2.

Fast kernel summation: Similar to the nearest neighbors problem, the direct calculation of Eqn. (2) requires $\mathcal{O}(n^2)$. For two and three dimensional case, fast multipole method (FMM) can reduce it to $\mathcal{O}(n)$ [22, 60]. However, FMM starts losing its effectiveness drastically as the dimensions is larger than three. In high dimensions, people have proposed several approaches that cost $\mathcal{O}(c^d)$ or $\mathcal{O}(d^c)$ where c is some constant. But lots of them depends only on the ambient dimension. For real applications, data often have a low dimensional manifold structure. The only work we know that depends on the intrinsic dimension is

Table 1: *Largest problem size we solved for each nearest neighbors search approach.* We listed the largest data size for each method in our experiments. Those experiments are all for the all-to-all neighbors search, which means the number of query points are equal to the reference points (i.e., $m = n$), except for the metric tree, where each process has 10,000 reference points but only 1,000 query points. Roughly speaking, the brute force approach can only process small size of data due to space and time limitation. The efficiency of metric tree for exact search primarily depends on the pruning percentage, which is determined by the intrinsic dimension of data. The accuracy of LSH and the RKD TREE is also based on the intrinsic dimensions.

data size	exact search			approximate search	
	rectangular	cyclic	metric tree	LSH	RKD TREE
total #points (million)	82	12	160	1639	819
grain size	5,000	1,000	10,000	50,000	50,000
#processors	16,384	12,288	12,288	32,768	16,384
intrinsic dim	1,000	100	5	10	10
ambient dim	1,000	100	100	2,048	2,048
million cycles/point/core	12	266	1	55	40

[77]. However, it does not construct a representation of the far points contribution, and it converges slowly. For more details, please refer to §3.1.

We present "ASKIT", a fast kernel summation treecode with a **new near-far** field decomposition and a new compact representation for the **far field**.

- *New near far decomposition:* We use a combinatorial criterion based on nearest neighbors for near/far field decomposition. It is easy to implement and essentially the percentile of pruning depends only on the intrinsic dimensionality of data.

- *New far field representation:* We represent far field contribution by using an approximate interpolative decomposition (ID) which uses nearest-neighbor information for sampling. The effectiveness only depends on the rank of sub-blocks of the kernel matrix and the compression efficiency depends on the intrinsic dimensionality of data.

- *Scales up to thousands of cores:* Our code is implemented in parallel and able to

scales to thousands of cores. On a 5M-point, 18D UCI dataset we obtain 25× speedup. On a 5M-point, 128D syntehtic dataset we obtain 2000× speedup using 256 dual-scocket nodes.

Modeling of likelihood and prior functions: Modeling likelihood and prior probability functions has generated lots of research interests including parametric models such as mixture of Gaussians [12], Gibbs random field [57], Markov random field [137], etc., and nonparametric models, for example, the kernel density estimation (KDE) [37]. Parametric approaches always assume the data are sampled from a certain distributions, for instance, Gaussian or Dirichlet distributions. However, there is often no way to know the real distribution of the sample data. If the assumption is far away from the real distribution, it is impossible to get accurate results. In order to approximate the underlining distribution of sampled data, nonparametric methods like KDE would approach to the real distribution at any accuracy as long as there are enough training data [120]. In standard practice, the likelihood and prior are computed by assuming that the pixels are independent and identically distributed (i.i.d.) or they nearby pixels are correlated in a certain way (random field). This assumption can be the best one can do in the absence of large enough training sets. We propose an alternative way to represent the probability function by KDE and provide preliminary experimental evidence of the performance of the new representation. In details,

- *New representation of likelihood function (chapter 4):* We demonstrate that the assumption that pixel intensities conditioned on the labels are i.i.d. is problematic and propose a framework for alternative representations.

- *New representation of prior function (chapter 5):* We study a new statistical spatial prior function which does not require preprocessing alignment. Out new spatial prior can be easily to be extended to many spatial inverse problems by adding it as a regularization to the main optimization.

- Leverage our work on a fast and parallel algorithms for high dimensional KDE and

provide experimental results to show that our presentation is robust to noise, allows full use of training images (and not only some averages), requires no initialization and manual interaction and is general and extensible.

Medical image segmentation using training set: Lots of research and literatures have been done for medical image segmentation. Please refer to §4.1 and §5.1 for a more detailed literature review. Roughly speaking, due to noise, artifacts, variations of structures and high accuracy requirement, to make use of training set and prior anatomical knowledge, we adopt a supervised Bayesian framework which optimizes the MAP in Eqn. (1) to segment a given image. It is easy to put our new representation of both likelihood function and prior function. The biggest challenge is the optimization procedure is usually time consuming and sometimes prohibitive.

- We proposed a fast and matrix free algorithm to optimize the MAP in parallel using TAO and PETSC.
- We use both the first and the second order derivatives of the object function to accelerate the convergence. The evaluation of gradient and Hessian can be implemented by Fast Fourier Transform (FFT) to reduce the computational cost significantly.
- The evaluation of KDE is based on our nearest neighbor search package in parallel and is capable to deal with large scales of training images.

1.4 Outline of the Dissertation

Each of the aforementioned contributions is individually presented in a chapter of this dissertation. In Chapter 2 we present the parallel nearest neighbors search algorithms, implementations as well as the scalability results. Chapter 3 outlines our ASKIT algorithm for the fast parallel approximation of kernel summation in high dimensions. Chapter 4 describes the modeling of likelihood functions and its ability in medical image segmentation.

Chapter 5 extends the bayes framework with new prior which gives a full MAP estimation, and the effectiveness of our new prior energy as a new regularizer in inverse problem.

CHAPTER II

CLUSTERING AND NEAREST NEIGHBORS SEARCH IN HIGH DIMENSIONS

In this chapter, we consider nearest neighbor searches in high dimensional space. Although there is a significant body of work for efficient nearest neighbor searches in database systems, surprisingly little work has been done on algorithms and libraries for high-end scientific computing platforms. In particular, libraries and algorithms that allow simulation codes to be coupled directly with data analysis for end-to-end parallelism and that scale to thousands of cores are absent. We present a set of basic tools built on top of Message Passing Interface (MPI) and OpenMP for massively parallel computational geometry problems. In particular, we combine parallel distance and filtering operations and kmeans clustering algorithms with an optimized seeding procedure, locality-sensitive hashing, and a novel parallel indexing tree data structure, "RKD TREE", to support, exact, and approximate basic operations for problems in high-dimensional computational geometry.

We outline the algorithms and provide preliminary performance benchmarks test. In our tests, the overall scheme shows excellent scalability and enables the analysis of datasets of unprecedented scale. In our largest runs, we were able to conduct exact searches and clustering on datasets with over four billion reference points in 1000 dimensions (a 45TB dataset) in under 30 seconds on 98K cores on the NSF NICS Kraken platform. As a second performance highlight, both tree construction and an all-nearest neighbors query take less than two minutes for a dataset with over 240 million points in ten dimensions on 6144 MPI processes. For 100-dimensional dataset the construction is still fast but due to limited pruning the query is four to five times slower.

2.1 Introduction

Nearest neighbor and clustering problems are fundamental problems in computational geometry. They are building blocks for more complex algorithms in computational statistics (e.g., kernel density estimation), spatial statistics (e.g., n-point correlation functions), and machine learning (e.g., classification, manifold learning). In turn, such methods are key components in physics (high-dimensional and generalized N-body problems), dimension reduction for scientific datasets, and uncertainty estimation Examples of the applicability of these methods to science and engineering include image analysis and pattern recognition [123], materials science [55], cosmological applications [59], particulate flow simulations [109], and many others [1]. As recent reports assert [2], it is particularly important to scale computational data analysis algorithms to allow end-to-end simulation and analysis and processing of very large datasets.[1]

Despite `KNN` and `KMC` methods being fundamental building blocks for many algorithms in computational data analysis, there has not been a lot of work in scaling them to high-performance parallel platforms. There is a rich literature in distributed memory algorithms, particularly in the database community, but the technologies have not yet been migrated to high performance computing platforms.

The kmeans clustering problem is NP-hard, even for two clusters, so only approximate algorithms are tractable. One such approximate method is the kmeans method, which provides a simple iterative scheme to compute the clusters. There are two issues with this method. The first one has to do with its sensitivity to seeding. Random seeding leads to very slow convergence and an excessive number of iterations. Furthermore, it requires repeated seeding, often hundreds of times to improve seeding. This can lead to excessive communication costs as every kmeans iteration involves an "`allreduce()`" operation.

[1] For example, in our blood flow simulations [109], the unprocessed raw data can reach 15 petabytes (for 300 million red blood cells and 10K time steps), which, with compressing and sampling, can be reduced to 100 terabytes.

The second issue is related to the cost of the overall calculation as the number of clusters increases. A straightforward implementation of the algorithm (described in **??**) has a computation cost that scales as $\mathcal{O}(kn/p + k \log p)$, where p is the number of MPI processes. Other techniques like hierarchical kmeans produce better quality clusterings, but their work scales as $\mathcal{O}(n^2)$, and they are harder to parallelize.

The k-nearest neighbors problem has a trivial $\mathcal{O}(nm)$ algorithm. If $m = n$ (the problem is also known as the *all nearest neighbor* or *ANN*), direct search is prohibitively expensive. Spatial data structures can deliver $O(n \log n)$ or even $O(n)$ complexity for fixed dimension d. But for *high dimensions* (say, $d \geq 4$), spatial data structures provably deteriorate, and for very large d all known exact schemes end up having the complexity of the direct algorithm [129].

In many applications however, the "signal" is concentrated in a lower-dimensional manifold. In such cases, *approximate searches* using indexing techniques (tree-type data structures or hashing techniques) can perform better than direct searches. Although there is a lot of work on algorithms and implementations of such methods for distributed databases, there is limited work on algorithms and parallel scaling analysis on high-performance computing applications. In this paper, we discuss a scheme that combines indexing and hashing.

The rest of this chapter is organized as follows: in the remainder of this section we outline our contributions in parallelizing `KNN` as well as the limitations of our work. Then we review the related work to the study of `KNN` in high dimensions and parallel implementation. In §2.2, we describe the randomized seeding and kmeans clustering algorithms. In §2.3 we introduce basic algorithmic components of the method:

- the single node optimization of distance calculation and `KNN` search (shared memory parallelizatoin),
- two schemes of direct parallel `KNN` search,
- the parallel locality sensitive hashing algorithm and implementation,

- the parallel tree construction and query algorithms,

In §2.4, we focus on the performance and scalability of the method on synthetic datasets and not so much on measuring the quality (for the approximate methods). We plan to apply our clustering and nearest neighbor search algorithms on image and fluid mechanics datasets later.

2.1.1 Our approach and contributions

we studied the performance of exact and approximate implementations as a function of the problem size, the number of dimensions. In particular,

- *Randomized seeding for kmeans*. We implemented and parallelized the randomized seeding and ball-clustering algorithm of Ostrovsky et al. [99]. This algorithm improves the standard kmeans algorithm by orders of magnitude as there is no need for iterative seeding. It has excellent scalability for small k, since it dramatically reduces the number of iterations. Its scaling for large k is part of ongoing work and will be reported elsewhere.

- *Direct nearest neighbors*. We propose two parallel algorithms for the direct calculation of the `KNN` problem. The first one trades memory in favor of computing time by replicating the query and reference points, minimizing the synchronization costs. For k-queries the work scales roughly as $\mathcal{O}(\frac{nm}{p} + k \log p)$ and its storage scales as $\mathcal{O}(mnp)$. The method is useful for small m or for the case in which wall-clock time is important. It is work-, but not memory-, optimal. The second algorithm uses a cyclic iteration whose time complexity scales as $\mathcal{O}(\frac{nm}{p} + m + n + \frac{mnk}{p})$ and the memory scales as $\mathcal{O}(nm)$. Both of these algorithms can be used to compute exact distances, verify correctness of approximate algorithms, and when m is small—such as in our fast `KNN` and `KMC` algorithms.

- *Parallel locality sensitivity hashing (*`LSH`*).* We parallelize the algorithm developed by Andoni and Indyk [5] for approximate searches. We present a distributed memory version of the algorithm, along with autotuning for parameter selection, memory and CPU optimizations.

- *Parallel tree construction and near-neighbor searches*. We present RKD TREE, a novel tree construction and traversal algorithm for indexing structures in arbitrary number of dimensions. Our method can be used for several types of trees (e.g., ball trees, metric trees, spill trees, or KD-trees). It is a recursive, top-down algorithm in which every node corresponds to *a group* of processes and one group of reference points. The key features of our method are that the tree is not replicated, it allows locality by minimizing the cost of collective communications, it is relatively easy to implement, and it can automatically switch to an approximate search when poor pruning performance is detected. The tree is used to partition and prune spatial searches across MPI processes. Once we reach granularity of an MPI process, we switch to multi-core exact or approximate algorithms. In our implementation, we use the locality-sensitive hashing scheme (`LSH`) [5] for approximate searches, which we also have implemented using shared and distributed memory parallelism.

- *Scalability tests on up to 240K cores*. We integrate single-core optimizations, shared memory parallelism (using OpenMP), distributed memory parallelism (using MPI), to enable calculations of unprecedented size, in terms of the dataset size (24 TB), number of dimensions (2,048), and number of cores (240K).

To give quantitative information about all the aspects of our algorithm, we also discuss the performance of the `LSH` method on a single socket and we give a limited snapshot of its performance for synthetic datasets. Also, a key component in our computations is the DGEMM kernel, which enables multithreading. In addition, we use OpenMP parallelism for sorts, scans, and other calculations within a single socket. *To our knowledge, our*

framework is the first scheme that enables such levels of performance and parallelism for arbitrary-dimension computational geometry problems.

2.1.2 Limitations

RKD TREE has several limitations: (**1**) It is not appropriate for frequent insertions or deletion of points. (**2**) For certain high-dimensional data sets in which the data points are not concentrated on a lower-dimensional manifold, RKD TREE is unable to prune points during a query operation. The inability to prune, however, can be detected easily during tree construction, allowing for a graceful fallback to a brute-force or approximate query. (**3**) We have only examined ℓ_2 distance metrics and many applications require other distance metrics. (**4**) Many algorithms, such as the fast-multipole method, require building interaction lists among various tree nodes. We do not discuss parallelization of such operations here. (**5**) The tree traversals are done exactly. Further approximation is possible using inexact pruning.

2.1.3 Related work

For a comprehensive review on clustering methods and computational geometry, see [66] and, for parallel kmeans, see [96]. In [70], the authors present an efficient kmeans algorithm based on a KD-tree data structure; the method was not parallelized. Our future plan is to combine RKD TREE with those methods to circumvent the adverse scaling of the clustering algorithm with the number of clusters. The basic scheme for parallelizing kmeans is simple and is discussed in [43] and [69].

There are plenty of works has been proposed to accelerate nearest neighbors search based on tree structures. The most frequently used algorithm is the KD-tree [51], which partitions the space according to one coordinate in each recursion. Fukunaga et al. [53] propose another tree structure which groups points by clustering points with k-means into k disjoint groups. If the metric is non Euclidean, ball-tree or metric-tree might provide better performance [24, 125, 97]. There approaches currently have only trivial query time

guarantees of $\mathcal{O}(n)$. As the dimensionality increases, they would lost their effectiveness quickly, and to find the correct nearest neighbors, we should visit almost all leaf nodes in a tree.

For high-dimensional spaces, there are often no known algorithms for nearest neighbor search that are more efficient than the simplest linear search. However, recently people has proven that it is still possible to obtain good efficiency if data points lie along a low-dimensional sub-manifold which is embedded into a high dimensional space [72, 107, 11]. Namely, the search complexity is $\mathcal{O}(\eta \log n)$, where η is determined only by the intrinsic dimensionality of data points [122, 113]. It is a good heuristic that for points which have a low dimensional embedding, the tree could be pruned, and only a few leaf nodes need to be visited.

Although there is significant amount of work on indexing structures in the database community (e.g., [10]), and on sequential algorithms [59, 110] for generalized N-body problems and tree data structures, we are not aware of any message passing interface-based scalable algorithms for multidimensional trees. Overall, for both kmeans and k-NN problems, the previous work has been limited to shared memory implementations and distributed memory with quite small number of processes.

The details of the `LSH` algorithm can be found in [5]. The original implementation of `LSH` can be found in [4]. Also, nearest neighbor algorithms using direct search on GPUs can be found in [56]. In [101], the authors present excellent analysis and implementation of the `LSH` algorithm on a GPU architecture.

2.2 Parallel KMeans Clustering with Randomized Seeds

In this section, we outline the point-grouping mechanism, the kmeans clustering. We use a standard kmeans clustering algorithm with the seeding introduced in Ostrovsky et al. [99]. Its main feature is that it reduces the number of iterations required to converge the kmeans algorithm and most importantly, removes the need for multiple invocation of kmeans with

different seeds. For well-clusterable cases (e.g., mixtures of well-separated Gaussians), the new clustering can be orders of magnitude faster than uniformly randomly selecting seeds among the input points because it eliminates the need for repeated seed selection.

Parallelizing kmeans is straightforward [43]. Algorithm 1 describes the scheme. It is

Algorithm 1 kMEANS(r, c, n, k)

1: **for** $j = 1 \cdots n$ **do**
2: For each point r_j, assign cluster membership by finding the closest centers c_i
3: $c_i = \sum r_j, \quad r_j \in$ cluster i $(V_i), \quad |V_i| = n_i$
4: n_i = ALLREDUCE(n_i)
5: c_i = ALLREDUCE(c_i)
6: $c_i = c_i/n_i$
7: **end for**

easy to see that the complexity per iteration is $\mathcal{O}(kd(\frac{n}{p} + \log p))$, where k is the number of clusters, n is the number of points, and p is the number of processors.

2.2.1 Seeding

To reduce the iterations of standard kmeans and obtain high quality clusters, seeds can be carefully selected. A choice of the initial centroids (*seeding*) with provable quality guarantees (under a quantitative assumption of clusterability) is discussed in [99]. Once the seeds have been computed, the ball-kmeans or the standard kmeans iteration can be used. The algorithm in [99] is based on the observation that k initial centroids that are far away from each other will belong to k different clusters. To find these points, we first oversample on probabilities based on the interpoint distances and then we eliminate *bad* seeds.

The oversampling step is summarized by Algorithm 2. First, we sample two initial seeds (denote as s_1 and s_2) according to probabilities

$$p_j^1 = \frac{\left(d^2(r_j, \bar{r})n + \sum_j d^2(r_j, \bar{r})\right)}{\left(2n \sum_j d^2(r_j, \bar{r})\right)} \tag{3}$$

$$p_j^2 = \frac{d^2\left(r_j, s_1\right)}{\left(\sum_j d^2(r_j, \bar{r}) + nd^2(\bar{r}, s_1)\right)} \tag{4}$$

where n is the number of points, $\bar{r}$ is the global mean of all points r_j, s_1 is the first selected seed, and $d(\cdot, \cdot)$ is the distance between two points.

Then, for the remaining points, the sampling probability is given by

$$p_j = \frac{d^2(r_j, s_*^j)}{\sum_{r_j} d^2(r_j, s_*^j)} \tag{5}$$

where $d(r_j, s_*^j)$ is the distance between point r_j and its nearest seed s_*^j, which has already been chosen. Finally, a randomly sampled point based on the probability from the unchosen set is used until a new seed is added. We repeat these steps until we have chosen k seeds.

Algorithm 2 ADDSEEDS(r, s, k)

```
if ISEMPTY(s) then
    sample the first two seeds s_1 and s_2
end if
if |s| ≥ k then return
d(r_j, s_*^j) = min_{s_i} d(s_i, r_j)
compute sampling probabilities p = {p_j} according to Eq. 5
s' = RANDOM_SAMPLE(r/{s}, p)
s ← s ∪ s'
ADDSEEDS(r/{s}, s, k)
```

After oversampling k' seeds where $k' > k$, We eliminate these seeds one by one to exclude those which generate bad clustering quality, until finally we have k centers. The seed elimination step is described in Algorithm 3. In more detail, we evaluate the clustering quality of $k' - 1$ seeds $\{s_i\}_{i=1}^{k'} \backslash s_j$ for each s_j, then delete the one with the lowest quality. The clustering quality is measured by the kmeans loss $\mathcal{L}$,

$$\mathcal{L}_k(X) = \frac{1}{|X|} \sum_{i=1}^{k} \sum_{x \in V_i} \|x - s_i\|^2 \tag{6}$$

where V_i is the Voronoi set of seed s_i. If the data is ϵ-separated, which implies $\mathcal{L}_k(X)/\mathcal{L}_{k-1}(X) \leq \epsilon^2$, [99] suggests oversampling $k' = \frac{2k}{1-5\sqrt{\epsilon}} + \frac{2\log(2/\sqrt{\epsilon})}{(1-5\sqrt{\epsilon})^2}$ points is sufficient to obtain good results. In practice, without any prior knowledge about the ϵ, it is possible to estimate it by using a small subset of the input data.

Algorithm 3 ELIMINATESEEDS(r, s, k)

if $|s| \leq k$ **then** return
for $i = 1 : |s|$ **do**
    $n_i = |V_i|$ for s_i
    $s^{(i)} \leftarrow s \backslash s_i$
    compute the kmeans loss $\mathcal{L}^{(i)}$ of $s^{(i)}$
    $T_i = n_i \mathcal{L}^{(i)}$
end for
$s^* = \min_s T_i$
$s \leftarrow s \backslash s^*$
for $i = 1 : |s|$ **do**
    find V_i for each s_i
    $s_i \leftarrow mean(V_i)$
end for
ELIMINATESEEDS(r, s, k)

2.2.2 k-Means seeding performance

We compare the performance of two types of seeding approaches for kmeans. The first one is the standard uniformly randomly selection of seeds among the input points; the other is the Ostrovsky seeding.

Experiments run on the synthetic data, a mixture of eight Gaussians with unit variance. Each center of these Gaussians is located on a vertex of a hypercube. Each run was repeated 100 times. Clustering quality is measured by the loss function Eqn. 6 and the variance ratio $\mathcal{V}$

$$\mathcal{V}(X) = \max \frac{\sum_{i=1}^{k} p_i \|c_i - c\|^2}{\sum_{i=1}^{k} p_i \frac{1}{|V_i|} \sum_{x \in V_i} \|x - c_i\|^2}, \tag{7}$$

where $p_i = \frac{\|V_i\|}{\|X\|}$ and c is the center of the whole dataset X. The higher the variance ratio, the better clustering quality is.

When the length of the hypercube edges is 6, the Gaussians are relatively well separated, it is clear that the Ostrovsky seeds results in a better clustering than random selected seeds. First, the loss function and variance ratio indicate better quality. Second, the standard deviations of these measures are much smaller than those of random seeds, which implies the Ostrovsky seeds are more stable and can be expected to always produce good clustering

Table 2: *Clustering Quality:* The clustering quality is measured by the kmeans loss, variance ratio, number of kmeans iterations (**iters**), and the clustering accuracy (**accuracy**). We use a mixture of 8 Gaussians. Here D is the distance between each pair of Gaussian centers. k indicates the number of target clusters (the best k is eight).

D	k	**seeding**	$\mathcal{L}(X)$	$\mathcal{V}(X)$	**iters**	**accuracy**
3	4	random	4. 11±0.061	1.15±0.032	11.28±3.64	45.81%±0.88%
		Ostrovsky	4.10±0.046	1.16±0.024	11.01±4.33	45.61%±0.63%
	8	random	1.95±0.160	3.57±0.330	11.03±3.75	87.12%±5.65%
		Ostrovsky	1.97±0.171	3.51±0.361	8.02±3.57	86.30%±6.01%
6	4	random	11.20±0.421	1.60±0.083	7.91±2.85	49.91%±0.06%
		Ostrovsky	11.09±0.008	1.62±0.002	6.41±2.69	49.93%±0.04%
	8	random	3.55±1.216	8.34±3.528	7.44±2.29	91.25%±7.13%
		Ostrovsky	2.16±0.368	12.69±1.203	3.17±0.88	99.50%±2.16%

results. Third, the Ostrovsky seeds require less iterations to converge. We also test the case of edge length 3. In this case, there is no significant difference between two types of seedings. Generally speaking, only the Ostrovsky seeding can be "safely" used without need for repeated seeding, which is expensive as it requires collective communication. On the other side, the Ostrovsky seeds could also be a good clusterability heuristic. The loss function and the variance ratio will vary slightly when k is close to the number of data's intrinsic groups. In Figure 4, it is clear that the slope of the Ostrovsky seeding curve changes sharply when $k = 8$, which is the number of Gaussians we used. As a counterpart, it is less pronounced using uniformly random selected seeds.

2.3 Parallel KNN Approaches

We have implemented and parallelized the randomized seeding and ball clustering algorithm, and several scalable methods for solving both the k-nearest neighbors and ρ-near neighbors problems. In the following sections, we discuss the different components and we provide weak and strong scaling results.

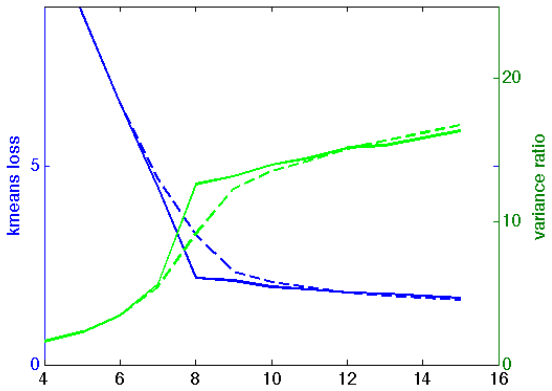


Figure 4: *Clusterability:* The change in the value of the loss and the variance ratio is an indication for the intrinsic number of clusters existing in the data. The blue lines stand for the kmeans loss and the green lines stand for the variance ratio. The solid lines are the Ostrovsky seeds, the dashed lines indicate random seeds. The x-axis is the number of clusters.

2.3.1 Single-Node Distance and k-Nearest Neighbor Kernels

Each of our k-NN and ρ-NN routines makes use of efficient single-node, multi-threaded kernels for distance calculations and/or k-NN searches on locally stored points. The distances between all (q_i, r_j) pairs of points are computed as follows. The sets of reference points $\mathcal{R}$ and query points $\mathcal{Q}$ are stored as $n \times d$ and $m \times d$ matrices, respectively, with 1 point per row. Then, we compute $D_{ij} = |q_i|^2 + |r_j|^2 - 2r_i \cdot q_j$ for all $(i, j), i \in 1 \ldots m, j \in 1 \ldots n$, where D_{ij} is the *square* of the distance between query point i and reference point j. By expressing the $-2r_i \cdot q_j$ term as the matrix-matrix product $-2\mathcal{R}\mathcal{Q}^T$, we are able to implement that calculation using a standard BLAS DGEMM call, which allows for high single-node performance and easy portability to various homogeneous and heterogeneous platforms.

With the above squared-distance kernel, a single-node k-NN calculation is quite simple. It can be implemented by calling the above distance routine and, for each query point, sorting the squared distances in ascending order while keeping track of the index of the reference point corresponding to each distance. Clearly, this approach exposes a great deal

of parallelism. However, for sufficiently small k, $nk < n \log n$, so we would waste a significant amount of time for small k by sorting each row of D in its entirety. Similar to the approach in [56], we address this problem by sorting the rows of D for large values of k and using a selection sort-like algorithm to select the k minimum distances for each query point using k linear scans of size n per query point for small k.

One implementation issue encountered in the development of the direct k-NN kernel involves the fact that vendor-tuned DGEMM routines provide very poor multi-threaded performance when multiplying a tall, skinny matrix by a relatively small matrix. In our code, this happens when computing distances between a small number of query points and a large number of reference points with low dimensionality. We have observed this problematic behavior in both Intel's MKL and GOTO BLAS.

2.3.2 Brute-force Direct k-NN and ρ-NN

First of all, we have implemented two distributed brute-force nearest neighbor algorithms according to different data partition schemes, i.e., two dimensional partitioning and cyclic partitioning.

2.3.2.1 Two-Dimensional Partitioning with Query Point Replication

We first consider a partitioning scheme in which the reference and query points are divided into r_{parts} and q_{parts} pieces, respectively, and distributed across $r_{\text{parts}} \cdot q_{\text{parts}}$ nodes (see Figure 5(a)). This scheme is more memory-intensive than the cyclic partitioning scheme we describe later since the reference points and the query points are replicated q_{parts} and r_{parts} times, respectively. However, all calculations performed are local until a reduction is performed at the very end.

Algorithm 4 shows how this partitioning is used to compute the k-nearest neighbors. Each process computes a matrix containing the distance between each (r_i, q_j) pair, and sorts each row of the matrix (the distances for each query point) to select the k minimum distances. Finally, we perform a k-min reduction among all processes which have the same

Algorithm 4 RECTDIRECTK($n_{\text{global}}, m_{\text{global}}, k, d, r_{\text{parts}}, q_{\text{parts}}$)

1: Choose n_{local} and m_{local}.
2: Read n_{local} reference points into r_{local} starting with $\lceil n_{\text{global}}/r_{\text{parts}} \rceil \cdot \text{id}$.
3: Read m_{local} query points into q_{local} starting with $\lceil m_{\text{global}}/q_{\text{parts}} \rceil \cdot \text{id}$.
4: D = computeDistances($r_{\text{local}}, q_{\text{local}}, n_{\text{local}}, m_{\text{local}}, k, dim$)
5: **for** $i = 0 \ldots q_{\text{parts}} - 1$ **do** Sort ith row of D
6: Perform k-reduction among processes with ranks $\text{id}\%q_{\text{parts}}, \text{id}\%q_{\text{parts}} + q_{\text{parts}}, \ldots, \text{id}\%q_{\text{parts}} + (r_{\text{parts}} - 1) \cdot q_{\text{parts}}$.
7: Process has k-nearest neighbors for q_{local} if id $< q_{\text{parts}}$.

query points.

Assuming the query and reference points are partitioned into an equal number of pieces, each process's memory consumption grows as $\mathcal{O}(\frac{n}{\sqrt{p}} + \frac{m}{\sqrt{p}} + \frac{nm}{p})$, since each set of points is replicated $\sqrt{p}$ times. Since our algorithm requires an all-pairs distance calculation, a sort of the computed distances (selection sort for small k and merge sort for large k), and a reduction on the k minimum of those distances, its time complexity is $\mathcal{O}\left(\frac{mnd}{p} + \frac{1}{\sqrt{p}}(m + n) + \frac{mn}{p}(\log \frac{n}{\sqrt{p}}) + k \log \sqrt{p}\right)$ for large k, and $\mathcal{O}\left(\frac{mnd}{p} + \frac{1}{\sqrt{p}}(m + n) + \frac{mn}{p} + k \log \sqrt{p}\right)$ for small k.

2.3.2.2 Cyclic Partitioning for Large Problem Sizes

By using a partitioning scheme that does not replicate any data across processes, we can solve the exact k-nearest neighbors for significantly larger problem sizes than would fit into a machine's memory when using the replicated partitioning scheme. However, the reduced memory footprint comes at the cost of additional computation time. As illustrated in Figure 5(b), in this method, both r and q are partitioned into p nearly-equally sized partitions distributed among the processes.

Algorithm 5 shows how to use such a partitioning to compute the k-nearest neighbors. The algorithm works as follows. A given partition of q remains pinned to its "home" process, while in each of p communication steps, each partition of r shifts one process to the left in a "ring" of processes. At each communication step, a process runs the local direct k-NN kernel and merges the results with the current k minimum distances for each query

	q_1	q_2	q_3	q_4
r_1	r_1q_1	r_1q_2	r_1q_3	r_1q_4
r_2	r_2q_1	r_2q_2	r_2q_3	r_2q_4
r_3	r_3q_1	r_3q_2	r_3q_3	r_3q_4

(a) Rectangular partitioning

iter\node	1	2	3	4
1	r_1q_1	r_2q_2	r_3q_3	r_4q_4
2	r_4q_1	r_1q_2	r_2q_3	r_3q_4
3	r_3q_1	r_4q_2	r_1q_3	r_2q_4
4	r_2q_1	r_3q_2	r_4q_3	r_1q_4

(b) Cyclic scheme

Figure 5: *Data partition for direct search.* (a) Diagram of rectangular partitioning. Here the query points are partitioned into four parts and the reference points into three. The processes in each column are part of the same group when the k-reduction is performed at the end of the algorithmh. (b) Diagram of rectangular partitioning. Here the query points are partitioned into four parts and the reference points into three. The processes in white are part of the same group when the k-reduction is performed at the end of the algorithm.

point.

The communication cost of this algorithm is trivially $\mathcal{O}(pt_S + ndt_T)$, where t_S is the setup time (latency) required for each message, and t_T is the time required to transmit each double-precision value. Because communication and computation can be overlapped completely for sufficiently large m and n, the time complexity is determined by the time spent in the local direct k-NN calculation, which is $\mathcal{O}(\frac{mnd}{p} + m + n + \frac{mn}{p} \log \frac{n}{p})$ for large k and $\mathcal{O}(\frac{mnd}{p} + m + n + \frac{mnk}{p})$ for small k. The memory cost for this approach is $\mathcal{O}(\frac{mn+m+n}{p})$.

We have also implemented an ρ-near neighbor search based on the cyclic pattern. However, we do not show pseudocode or present performance results because of its similarity to the k-nearest neighbors search algorithm.

2.3.3 Locality Sensitive Hashing

Locality Sensitive Hashing (`LSH`) is a means of using one or more hash functions to map points to integral values such that two points x_1 and x_2 will–with known probability p_1– hash to the same value if the distance between them is less than a given threshold. It is equally important to bound the probability, p_2, that x_1 and x_2 have a false collision (i.e., that they map to the same value when their distance is "large"). For any given hash function

Algorithm 5 CYCLICDIRECTK($r, q, n_{\text{global}}, m_{\text{global}}, k, dim$)

```
1: Choose send_partner and recv_partner.
2: for i = 0 ... p − 1 do
3:    if kmin_set then
4:       kmin_new ← directKQuery(r, q, n_local, m_local, k, d)
5:       temp ← kmin_merge(kmin, kmin_new, m_local, k)
6:       kmin ← temp
7:    else
8:       kmin ← directKQuery(r, q, n_local, m_local, k, d)
9:       kmin_set ← TRUE
10:   end if
11:   send(r, send_partner)
12:   r ← recv(recv_partner)
13: end for
14: send(r, send_partner)
15: r ← recv(recv_partner)
```

to be useful, it must hold that $p1 > p2$. In our `LSH` implementation, we use the family of hash functions used in [5],

$$h_i(x) = \left\lfloor \frac{(\mathbf{a}_i \cdot \mathbf{x} + b_i)}{W} \right\rfloor, \tag{8}$$

where x is a d-dimensional data point, a is a d-dimensional vector with entries drawn from a 2-stable random distribution and b is a scalar drawn from $\mathcal{U}[0, W]$. Equation 8 projects each point x to the real line and breaks that line into segments of length W. Any points within the same line segment will be assigned the same hash key.

However, with the above hash family, the difference in probabilities p_1 and p_2 is too small. This difference is magnified by using $\mathcal{V}$ hash functions, and concatenating the resulting values to create the point's key. That is, given a point x, $h'(x) = (h_1(x), h_2(x), \ldots, h_{\mathcal{V}}(x))$. Since the overhead of storing, comparing, and (in the distributed case) transmitting $\mathcal{V}$ integer keys for each point can be quite high for large $\mathcal{V}$, we compress $h'(x)$ to a single (unsigned) integer value as follows: First, let $\mathbf{u} = (u_1, u_2, \ldots, u_{\mathcal{V}})$ be a vector of uniformly distributed random integers. Then, we calculate $h''(x) = (\mathbf{u} \cdot h'(x)) \mod (2^{32} - 5)$ ($2^{32} - 5$ is the largest prime number that can be represented in 32 bits). The random values in the vector $\mathbf{u}$ are necessary to reduce the number of false collisions caused by the compression

as the values $h_j(x)$ may all be confined to a relatively small range. Two points x and y are stored to the same bucket–and thus, included in a direct evaluation–if $h''(x) = h''(y)$. Thus, a higher value of $\mathcal{V}$ makes the hash function more selective. To reduce the likelihood of false negatives, we use multiple ($\mathcal{L}$) independent hash tables, each with its own set of $\mathcal{V}$ functions. See [40] for additional details on the `LSH` scheme used in both our library and `E`^{2}`LSH`.

Typically, `LSH` would be used as follows:

1. Generate $\mathcal{V}\mathcal{L}$ hash functions as described above.

2. Compute $\mathcal{L}$ hash values $h''_i(r)$ for each point in $r \in \mathcal{R}$, using the function set for hash table T_i.

3. Allocate $\mathcal{L}$ hash tables, and insert the ID of point r into bucket $h''_i(r)$ of table T_i.

4. Compute $\mathcal{L}$ hash values $h''_i(q)$ for each point in $q \in \mathcal{Q}$, using the function set for hash table T_i.

5. For each query point q, return all point IDs in $\bigcup_{i=1}^{\mathcal{L}} T_i(h''_i(q))$, where $T_i(a)$ is bucket a of hash table T_i.

During the first phase, the data points are mapped to buckets using Equation 8, and during the second phase, this same process is used to map the query points to their buckets. Then, the data points already in the buckets are reported as near neighbors. A k-nearest neighbors search can be performed over the neighbors reported by `LSH` in order to eliminate false positives and report the k-nearest data points.

Our implementation scales `LSH` to high-performance shared- and distributed-memory architectures. It further differs from conventional `LSH` implementations in that points are stored in contiguous arrays, grouped by their hash keys, rather having their IDs inserted into a hash table. While sorting points by key is more expensive than n inserts into a

traditional hash table data structure, this operation is easily parallelized in a multi-core system, whereas hash table inserts are very difficult to parallelize efficiently. Additionally, the sorted-table approach substantially reduces the number of memory accesses required in the query phase since the multilevel table representation proposed in [40] involves following several pointers. Additionally, recall that in the original representation, the points themselves are not stored in the tables, so all points within a bucket must be gathered to a contiguous array for evaluation with a BLAS-based kernel. Experimentally, we find that a negligible increase in construction time (less than 40% for 100,000 points) results in reduction in query time of up to three orders of magnitude for large query point sets ($m = n$).

While `LSH` schemes in general solve only the approximate ρ-near neighbor problem directly, by selecting the k nearest ρ-approximate neighbors returned for each query point, one can obtain an approximate solution for the k-nearest neighbor problem as well. However, it is still necessary to use a ρ that is "reasonably" likely to contain the k nearest neighbors of each query point. In this section, we focus on the *approximate k-nearest neighbor problem*. That is, given a query point q and a set of reference points $\mathcal{R}$, we seek to return a vector of k neighbors $\nu = (\nu_1, \nu_2, \ldots, \nu_k)$ such that the distance between q and each ν_i is *similar* to the distance between q and the corresponding member of the exact nearest-neighbor vector ν'. In other words, we want to find k neighbors such that the *mean relative distance error* E_q for query point q, $E_q = \frac{1}{k}\sum_{i=1}^{k} \frac{||\nu_i - q| - |\nu'_i - q||}{|\nu'_i - q|}$, is small. In the case where the query point set $\mathcal{Q}$ contains m query points ($m > 1$), we define $E = \frac{\sum_{i=1}^{m} E_{q_i}}{m}$. While we can guarantee with fixed probability that the neighbors are correct for *the majority* of query points (see *Parameter Selection* below), we cannot provide a theoretical error bound for those points whose k-nearest neighbors lie outside of the search radius. However, we find that in practice, a very low value of E (less than 0.01) is attainable even on highly non-uniform synthetic data sets.

2.3.3.1 Distributed LSH Algorithm

While distributed-memory algorithms and implementations of LSH are not new (and in fact, many based on the same hash function family exist), all such algorithms of which we are aware suffer from limitations that make them unsuitable for HPC applications. For example, the authors of [127] present a highly scalable indexing scheme based on LSH. However, their method provides no support for bulk queries. Lv *et. al* propose the "multi-probe" LSH method, [86] which is also derived from the scheme presented in [40]. In this scheme, each query point visits multiple buckets in each table, which reduces the number of tables needed. However, the authors find that this increases the overall query-phase time by 30%-210%. Therefore, it is questionable whether the reduction in $\mathcal{L}$ would provide any real performance benefit, except on the slowest of networks. In [63], an algorithm is described which distributes a multi-probe index over the nodes of a peer-to-peer network. However, this method also is limited to individual query points as the workload of a bulk query would be impossible to balance given the presented load-balancing scheme. The author of [102] presents another distributed implementation of the multi-probe scheme. However, in this scheme, each process builds an index of its local reference points, and those indices are queried using a simple master-worker architecture. Since all results are collected at the master process, the scalability of this algorithm is inherently limited. Our implementation supports massive reference and query point sets, exhibits good strong- and weak-scaling to thousands of cores, and is portable to a wide variety of high-performance parallel architectures. Because the error properties of the underlying LSH scheme are well established, we focus our discussion primarily on performance and scalability, rather than accuracy.

Our distributed LSH algorithm works similarly to the original sequential algorithm, except that the hash tables are distributed across all MPI processes. While we do allow the largest buckets to span multiple processes to allow for a good load balance of the direct evaluations within buckets, care must be taken to choose a sufficiently large $\mathcal{V}$ to prevent

extremely large buckets as these make the workload impossible to balance. Of course, a larger $\mathcal{V}$ also implies that a larger $\mathcal{L}$ is needed to provide sufficient accuracy. Because $\mathcal{L}$ can be relatively large for very large problem sizes ($\mathcal{O}(100)$), it is not feasible to store all $\mathcal{L}$ hash tables in memory. To address this issue, we perform $\mathcal{L}$ iterations in which a single table is constructed and and queried. Although each of these iterations incurs a significant communication cost from multiple all-to-all exchanges, we still achieve good performance and scalability compared to a direct search. Because it is possible that some of the randomly generated hash functions will produce "bad" tables (i.e., tables with workloads that are difficult to balance), we verify that the workload can be balanced without assigning more than $\frac{p}{2}$ processes to multi-process buckets and discard the current hash function if this cannot be done. The number of processes assigned to large buckets must be limited because the number of processes assigned is proportional to the quadratic computational work, rather than the linear memory cost. That is, a bucket's workload may be orders of magnitude greater than the next-largest bucket while its point count may be only a factor of 2-3 greater. This can lead to very few processes being available to store small buckets' points if we don't impose such a limit. In practice, discarding "bad" tables maintains a good load balance without appreciably impacting accuracy, provided that the number of discarded functions is not significantly greater than $\mathcal{L}$. Excessive discards can always be avoided with a good choice of $\mathcal{V}$.

Algorithm 6 outlines the operation of the distributed `LSH` k-nearest query. A ρ-near query would require only trivial modifications to the algorithm but is not currently implemented. As discussed above, however, a ρ parameter is still required for the approximate k-nearest search. We omit details related to discarding hash functions. It is important to note that the algorithm as presented is efficient *only for large query point sets*. In the case of a small number of query points (i.e., small enough to fit within a single process's memory), each process could run the same algorithm locally on its partition of the reference point set. The results could then be collected with a "k-minimum" reduction.

Algorithm 6 Distributed LSH k-Query

1: DIST_LSH($r, q, n, m, \mathcal{V}, \mathcal{L}, \rho, k, \beta$)
2: if(rank==root) hashPool = GENERATEHASHFUNCTIONS($\mathcal{V}, \mathcal{L}, \rho$);
3: BCAST(hashPool)

Hash r and check for excessive bucket size.
4: **for** $l = 0 \ldots \mathcal{L} - 1$ **do**
5: refKeys = COMPUTEHASHKEYS(r,hashPool[$\mathcal{V}l$], β)
6: refKeyCount = LOCALBUCKETSIZES(refKeys)
7: REDUCE(refKeyCount, SUM)
8: queryKeys = COMPUTEHASHKEYS(q,hashPool[$\mathcal{V}l$], $p\beta$)
9: queryKeyCount = LOCALBUCKETSIZES(queryKeys)
10: REDUCE(queryKeyCount, SUM)

Calculate the amount of work required to evaluate each bucket.
11: $\mathcal{W}$ = WORKPERBUCKET(queryKeyCount,refKeyCount)
12: bucketMap = LOADBALANCE($\mathcal{W}$) *//Assign buckets to processes*

Transmit points to assigned procs.
13: REDISTRIBUTEPOINTS(r, refKeys, bucketMap)
14: REDISTRIBUTEPOINTS(q, queryKeys, bucketMap)
15: rBuckets = GROUPBYKEY(r, refKeys)
16: qBuckets = GROUPBYKEY(q, queryKeys)
17: **for** each $b_q \in$ qBuckets **do**
18: new_results = new_results $\cup$ DIRECTKQUERY(b_r, b_q)
19: **end for**

Group results by ID and transmit to each query point's "home" process.
20: GROUPBYID(new_results)
21: new_results = REDISTRIBUTERESULTS(new_results)
22: results = KMIN(new_results, results) *//k nearest of new and existing results for each query point*
23: **end for**
24: **return** results

Each of the $\mathcal{L}$ iterations proceeds as follows: The hash keys for the reference points are computed and compressed to the interval $[0, \beta)$ Here, β is the total number of hash buckets. The number of buckets must increase with p to allow for a good load balance[2]. This compression step is necessary because we must store an array mapping the keys to MPI ranks. We then find the size of each bucket (the number of occurrences of each key). We assign buckets to processes such that each process has approximately equal work, store the bucket-to-process mapping, and redistribute the reference points to the newly assigned processes.

Next, we compute and compress the hash keys of the query points and send them to the processes indicated by the bucket-to-process map. Finally, we perform the exact `KNN` search on the sets of reference and query points with colliding hash keys, transmit the new results for each query point to that query point's "home" process, and merge the new results with the existing results, keeping the k minimum. This entire process repeats for $\mathcal{L}$ iterations. Since the cost of each iteration is dominated by the direct distance evaluations and all-to-all personalized exchanges, the approximate complexity of this algorithm is $\mathcal{O}\big(\mathcal{L}[\frac{mn}{p^2} + pt_s + (m+n)d/pt_w]\big)$ in the average case, where t_s is the latency cost of each message, and t_w is the time required to transmit each double-precision number.

Load-balancing Direct Evaluations Because of the quadratic complexity of the direct evaluations within each bucket, it is necessary to estimate the relative evaluation time of each bucket and assign them to processes in such a way that each process's workload has roughly equal total cost. However, achieving a good load balance in practice is made more difficult by two facts: **(1)** the distribution of bucket sizes tends to be highly non-uniform, and **(2)** the quadratic cost estimate is very inaccurate for very small buckets, especially when OpenMP parallelism is used.

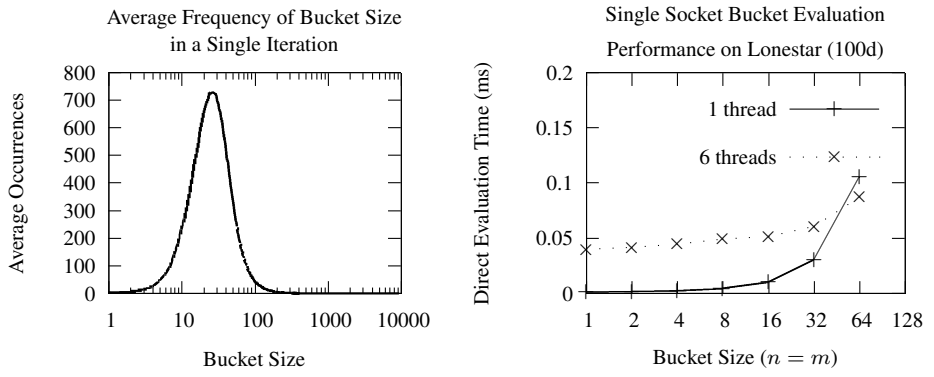
Figure 6(a) clearly illustrates problem (1). In this sample run on a 100 dimensional

[2]Up to a point, larger β values result in an improved load balance at the cost of higher memory consumption. As a guideline, β should be *at least* $\max(1000, 100p)$, but a generally good value is $\beta = 1000p$.

gaussian data set of 1.6M points and 32,000 total buckets, the largest bucket contains roughly 10,000 points, whereas the mean bucket size is approximately 25 points, and the vast majority of buckets contain fewer than 100 points. If we estimate $\mathcal{W}_i$, the workload of bucket i, to be $m_i n_i + m_i + n_i$–where m_i and n_i are the number of query points and reference points in bucket i, respectively–then the most expensive bucket is roughly 160,000 times more expensive than the "typical" bucket. If each bucket were assigned to only a single process, such a workload would be impossible to balance. Accordingly, we determine the ideal workload $\mathcal{W}'$ for each process and consider a bucket i to be "large" if $\mathcal{W}_i > \mathcal{W}'$. We then assign bucket i to round($\mathcal{W}_i$) processes. Bucket i's reference points are replicated across all processes assigned to it; its query points are evenly divided among the processes. We include the additive terms in the cost model to reflect the memory and bandwidth cost of each nonempty bucket, regardless of whether any evaluations are necessary.

The second challenge related to load balancing is the difficulty of accurately estimating the runtime of very small buckets using our simple $\mathcal{W}_i = m_i n_i$ cost model. Figure 6(b) illustrates this inaccuracy. While the evaluation times of large buckets follow the expected quadratic trend, very small buckets (i.e., those with a workload of roughly 32^2 or less) all end up requiring roughly the same amount of time to evaluate. This is primarily because the useful work to be done is not sufficient to amortize the overhead of multiple BLAS calls and the dispatching of work to OpenMP worker threads. Since the majority of buckets are likely to have a "very small" workload (see figure 6(a)), the underestimation of the cost of these buckets leads to a significant load imbalance. To combat this effect, our implementation merges any "very small" buckets into new buckets with workloads of at least 32^2. Since this also happens to be roughly the size where OpenMP parallelism realizes a speedup over sequential execution, there is added benefit to merging small buckets for hybrid MPI+OpenMP runs. While the exact behavior of small-bucket evaluations will vary with each platform, BLAS implementation, and OpenMP runtime, this results in greatly improved load balance and much faster overall evaluation times on tested systems. Because

the minimum cost of evaluating a bucket is machine-dependent, auto-tuning the threshold value may result in better performance on a wider range of systems.



(a) Histogram of Hash Bucket Sizes (b) Small Bucket Evaluation Performance

Figure 6: Left: This plot shows the average number of buckets per iteration of a given size. Note that the x-axis scale is logarithmic. The data were obtained by running a query on a 100-d gaussian data set with a total of 1.6M points and 32000 hash buckets. A similar distribution occurs for all *high-dimensional* data sets tested. **Right:** This graph shows the time (in milliseconds) required to evaluate buckets containing varying numbers of points on a single socket of the Lonestar platform. Each bucket contains an equal number of query and reference points.

Parameter Selection The performance of the `LSH` scheme we employ in our distributed query is known to be highly sensitive to the value of the parameter $\mathcal{V}$. In addition, both $\mathcal{V}$ and $\mathcal{L}$ must vary as a function of the input size, and (to ensure a given accuracy) $\mathcal{L}$ must be chosen as a function of $\mathcal{V}$. Furthermore, the accuracy of a k-nearest query depends on the value of ρ. Since the optimal value of each of these parameters is data-dependent, selecting them manually is not practical. For that reason, our implementation includes a function to quickly and automatically select good values for each of these three crucial parameters.

To select the parameter ρ, we be select a sample of $\mathcal{O}(\log m)$ query points uniformly at random. These sampled points are then replicated to all MPI processes, where a direct k-nearest search is performed against each process's locally stored reference points. Using our k-minimum reduction routine, we collect the results at rank 0 where we compute the mean μ and the standard deviation σ of the distances to the kth neighbors of all sampled query points. We then set $\rho = \mu + \sigma$, and broadcast ρ to all other ranks. Because of the

fact that the distances to the kth neighbors of query points can have a very high variance for non-uniform data sets, we have found that simply using either the mean or median is insufficient in general.

Determining the optimal value of $\mathcal{V}$ is a bit more difficult. A higher value of $\mathcal{V}$ will result in less computational work being done in each iteration; however, it will also increase the number of iterations required for a fixed level of accuracy (and hence, the communication cost). The reason for this is simple: a low $\mathcal{V}$ will produce larger buckets. Since this will result in more candidate neighbors being evaluated for each query point in every iteration, it is obvious that fewer iterations will be necessary to get a reasonably accurate result. We have observed that in general, there is an optimal per-bucket workload that will result in the best balance of communication and computation cost and, hence, the shortest overall running time. This optimal workload does not depend on the data set; it is a property of a particular parallel machine architecture. Therefore, this value need be determined only once for a target platform by simple benchmarking. We have found that buckets (both reference and query) of size 200 work well on a typical InfiniBand-interconnected cluster, but the optimal bucket size S could vary from roughly 100-1000, depending on a system's communication/computation balance.

Our $\mathcal{V}$-tuning algorithm requires a target bucket size parameter $\dot{S}$ and an initial value for $\mathcal{V}$, $\mathcal{V}_0$. As discussed in [40], we set $\mathcal{V}_0 = \lceil \log_{1/p_2} n \rceil$. In our implementation, we use $p_2 = 0.61$, which was chosen to minimize the probability of excessively large buckets. We also define the target workload $\dot{\mathcal{W}} = \dot{S}^2$, since it is the amount of *work*[3] in a bucket that we seek to optimize, rather than the number of points. We then search the range $[1, 2\mathcal{V}_0]$ using bisection, hashing all reference and query points 5 times for each value of $\mathcal{V}$ considered. We then compute the mean workload, $\overline{\mathcal{W}}$, of all trials of the current $\mathcal{V}$, excluding empty

[3]Note that here we omit the additive terms used in the load balancing cost approximation. Those terms primarily exist to ensure that every non-empty bucket has a non-zero cost, regardless of whether any direct evaluations will be performed. Since all non-empty buckets consume memory and network bandwidth, workless buckets cannot be ignored when load balancing, but they may safely be ignored here.

buckets from the calculation. Once the optimal value $\mathcal{V}_{opt}$ is found, we set $\mathcal{V}$ to be the nearest value (always $\geq \mathcal{V}_{opt}$) that does not produce an excessive amount of work in large buckets. This allows us to limit the number of discarded iterations when executing the query.

Finally, we choose $\mathcal{L}$ as described in [4]. That is, we define t to be smallest natural number that satisfies $(1 - p_1^{\mathcal{V}/2})^t + tp_1^{\mathcal{V}/2}(1 - p_1^{\mathcal{V}/2})^{t-1} < 1 - P$, where P is the specified probability of correctness. Then, $\mathcal{L} = \max(\lceil \frac{t(t-1)}{2}, \mathcal{V}) \rceil$. We set $\mathcal{L}$ to at least $\mathcal{V}$ because we find that $\mathcal{L}$ is otherwise insufficient for very small $\mathcal{V}$. Since we draw our hash functions from a Gaussian distribution, we use $p_1 = 0.8504$, which is determined by the formula given in [40].

Limitations: The distributed `LSH` algorithm has two limitations:

- It uses a single search radius for all query points. While this may work well for many data sets and applications, it can be problematic if accurate results are needed for query points that are distant outliers. However, if all reference points are very distant from a query point, it is unlikely that finding its k nearest neighbors would be meaningful.

- While we can guarantee with fixed (and user-specified) probability that neighbors found to lie within distance ρ of a query query point are correct, we cannot provide any such guarantee on the accuracy of more distant neighbors.

2.3.4 Random Projection Tree Search

Our goal in developing the tree data structure is to enable fast parallel pruning for near-neighbor query operations. In details, we arrange reference points into different groups in term of leaf nodes in a tree. Query points only need to visit part of them to find their near-neighbors. By no means do we claim that our method resolves the "curse of dimensionality" problems in general.

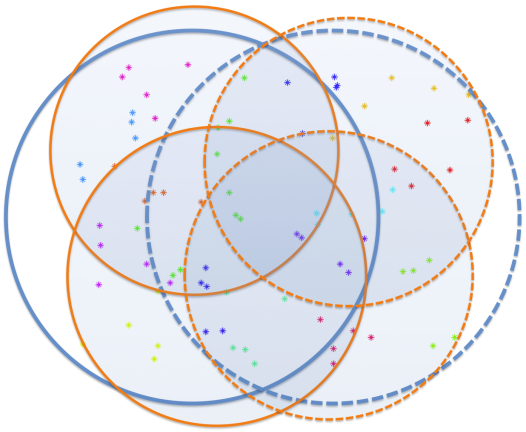
Most of the tree algorithms for high dimensions follow a top-down approach in which

the points are grouped recursively into tree nodes and then, during search, simple to sophisticated pruning strategies are used [114]. We follow the same strategy. Parallelizing tree operations with performance guarantees depends on the application and is hard even in low dimensions, especially in distributed memory architectures. Space-filling curves [75], graph partitioning [42] and geometric partitioning [89] have been used for parallelization, but for high-dimensional datasets these are not as successful and we are not aware of any parallel implementations. For this reason we have opted for a top-down basic approach.

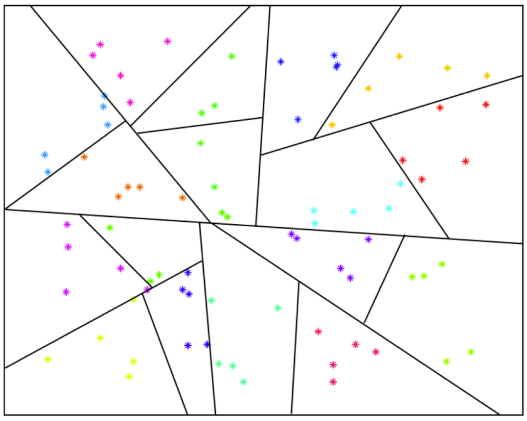
Before discussing the construction of the trees, we briefly compare two general points partition strategies: the hyperplane partition, and the clustering partition. Of course, other data structures could be used. The clustering partition applies kmeans to cluster points into k disjoint regions, at each level, when we split a node to two children, each of which contain one or more clusters. Whereas, the hyperplane partition always seek a hyperplane on the data space and then separate points into two sides. Points from each side form a child node of the tree. Expect for this, the other procedures are the same, as illustrated in Algorithm 7.

We prefer the hyperplane partition due to three main reasons. Firstly, partitioning points by hyperplane results in a perfect load balance for reference points over all processes. In other words, each child node has the same number of points as the others. It is important for further pruning and direct nearest neighbors search, and promise every process has a similar amount of work. Secondly, clustering based partition usually take more time to build a tree. If we only iteration a few times in kmeans to reduce the construction time [93], it is better to choose the initial seeds carefully to promise good clustering quality, which is also time consuming. Besides, the clustering based partition usually has overlap among the partition space, which may yield worse further pruning, even if the data points has a low intrinsic dimensionality. Figure 7 illustrate these two partition methods. Roughly speaking, there are usually overlaps between clusters at the same level, which means if you want to search for neighbors within in those overlapped points, you might need to visit both

of these clusters, which cannot prune very well and result in bad performance.



(a) clustering partition



(b) hyperplane partition

Figure 7: *Points splitting strategies.* An illustration of two points partition approaches. Usually the clustering partition results in overlap among different groups. If the dimensionality of data is high, it could be so serious that the radius of cluster would not shrink or shrink only a little compared to the clusters on their parent's level.

2.3.4.1 Tree construction

A standard tree construction uses the NODESPLIT function summarized in Algorithm 7. Let $\mathcal{T}$ be a tree node; $X_{\mathcal{T}}$ be all the points assigned in $\mathcal{T}$, $l_{\mathcal{T}}$ be the level of the node $\mathcal{T}$ in the tree, and $s_{\mathcal{T}}$ be the size of node $\mathcal{T}$'s communicator. Let $p_{\mathcal{T}}$ be the direction along which $X_{\mathcal{T}}$ are projected, and let $m_{\mathcal{T}}$ be the median of the projected values of $X_{\mathcal{T}}$, i.e., $m_{\mathcal{T}} = median(p_{\mathcal{T}} X_{\mathcal{T}})$. Denote $\mathcal{C}_l$ and $\mathcal{C}_r$ as the left and right child node of $\mathcal{T}$ respectively.

Algorithm 7 NODESPLIT($\mathcal{T}, X_{\mathcal{T}}$)

1: **if** $l_{\mathcal{T}}$ ==maxLevel || $|X_{\mathcal{T}}| <$ minNumofPoints **then**
2: store $X_{\mathcal{T}}$ in $\mathcal{T}$
3: return
4: **end if**
5: $p_{\mathcal{T}} =$ SPLITDIRECTION($\mathcal{T}, X_{\mathcal{T}}$)
6: $pX_{\mathcal{T}} =$ PROJECT($X_{\mathcal{T}}, p_{\mathcal{T}}$) // $px = x \cdot p_{\mathcal{T}}$
7: $m_{\mathcal{T}} =$ MEDIAN($pX_{\mathcal{T}}$)
8: $\{X_l, X_r\} =$ POINTSPLIT($X_{\mathcal{T}}, pX_{\mathcal{T}}, m_{\mathcal{T}}$)
9: NODESPLIT(LEFTCHILD($\mathcal{T}$), X_l)
10: NODESPLIT(RIGHTCHILD($\mathcal{T}$), X_r)

We test three ways to choose the split (projection) direction. The first one is selecting a coordinate axis at random, i.e., uniformly sampling a number from d integers. The second is to choose the coordinate with the largest variance considering all points within a node. Another alternative is to choose the direction where the points are furthest apart. To do this, we first compute the centroid c of all points within a node, then find the furthest point p_1 from c, and a second point p_2 which is furthest from p_1. Finally we project points along the direction $p = p_1 - p_2$. After projecting points onto this direction and calculate the median m of all projected values, we simply separate points into two groups by assigning points whose projected values is smaller than m to the left child, and the other to the right child. All of these three resulted in similar accuracies. For the rest of this paper, the complexity estimates are computed using the first version.

One way to parallelize Alg 7 is the one we described in [91] and uses recursive communicator splittings, which is illustrated in Figure 8.

As outlined in Algorithm 8, we use a distributed top-down algorithm to construct the tree and, at each level, split a node to two children, each of which contain one groups of points (in this implementation points locate on the lefthand side or the righthand side of the hyperplane). A tree node is shared among multiple processes. Each of those processes maintains a local data structure for the tree node containing its MPI communicator and the clustering information for pruning queries. In each process, these data structures are stored

Algorithm 8 PARNODESPLIT($\mathcal{T}, X_\mathcal{T}, C_\mathcal{T}$)

1: **if** $l_\mathcal{T}$ ==maxLevel || $|X_\mathcal{T}|$ < minNumofPoints **then**
2: store $X_\mathcal{T}$ in $\mathcal{T}$
3: return
4: **end if**
5: $p_\mathcal{T}$ = PARSPLITDIRECTION($\mathcal{T}, X_\mathcal{T}$)
6: $pX_\mathcal{T}$ = PROJECT($X_\mathcal{T}, p_\mathcal{T}$) // $px = x \cdot p_\mathcal{T}$
7: $m_\mathcal{T}$ = PARMEDIAN($pX_\mathcal{T}$)
8: $\{X_l, X_r\}$ = POINTSPLITREPARTITION($X_\mathcal{T}, pX_\mathcal{T}, m_\mathcal{T}$)
9: $\{C_l, C_r\}$ = COMMSPLIT($C_\mathcal{T}$)
10: PARNODESPLIT(LEFTCHILD($\mathcal{T}$), X_l, C_l)
11: PARNODESPLIT(RIGHTCHILD($\mathcal{T}$), X_r, C_r)

in a doubly-linked list representing a path from root to leaf for that process's portion of the tree. In our implementation, the minimum granularity of a RKD TREE node is one MPI process. At the leaf, we use an exact search kernel to find neighbors.

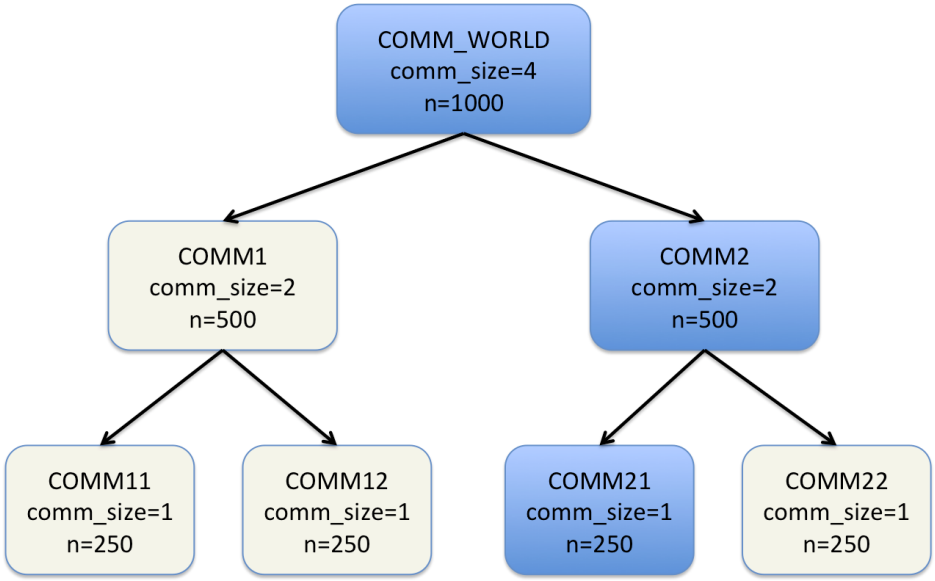


Figure 8: *Recursive tree construction and the corresponding communicators.* An illustration of the splitting of communicators used for reference point redistribution among children. The highlighted nodes represent a path from root to leaf as stored by a single process.

Finally we discuss the complexity of the tree construction algorithm in more details:

Choosing the split direction requires a broadcast, the projection is local, the parallel median (using randomized quick select) has an expected complexity of $\mathcal{O}(t_c \log p \log n + n/p)$, where $t_c = t_s + t_w$. Partitioning the points to the two communicators is the most communication-intensive part. In POINTSPLITREPARTITION(), the points are shuffled among processes, so that MPI processes that will end up in C_l have points only from the X_l (X_r for processes in C_r) subset.

For load balancing, it is necessary to ensure the workload is evenly distributed in the process of C_l and C_r. There are different ways to obtain the load balance, we present three variants for the POINTSPLITREPARTITION() function.

Variant A: We repartition the points by using `MPI_Alltoallv()`. Before calling this function, it is necessary to determine the process that each point belongs to, and ensure after repartition, each process has equal number of points. The algorithm that supports arbitrary spatial trees is given in Algorithm 9

Algorithm 9 GROUPDISTRIBUTE($\mathcal{T}, X_\mathcal{T}, C_\mathcal{T}$)

1: MPI_Scan(w_g, sw_g)
2: shift = 0
3: **for** $j = 1 : n_g$ **do**
4: shift $+=$ $(j == 1)$? 0 : group_size$[j-1]$
5: avg[j] = MPI_Allreduce($w_g[j]$) / group_size$[j]$
6: **for** i = 1 : n **do**
7: $\text{pid}(i) = (\text{membership}(i)! = j)$? $\text{pid}(i)$: shift $+ \lfloor (sw_g[j] + i - 1)/\text{avg}(j) \rfloor$
8: **end for**
9: **end for**

Assuming average message size of $\mathcal{O}(nd/p^2)$ Then the complexity of the exchange is $\mathcal{O}(t_w nd/p + t_s p)$. Therefore the expected complexity (omitting $\mathcal{O}()$) of one split at level ℓ with $p_\ell = p/2^\ell$ MPI tasks and n_ℓ points, then the **projection** costs $n_\ell d/p_\ell$ work, the median calculation costs $t_c \log^2 p_\ell \log n_\ell + n_\ell/p_\ell$ and the exchange $t_w n_\ell d/p_\ell + t_s p_\ell$. Median-based splits result in perfect load balancing and $n_\ell/p_\ell = n/p$. Summing over the levels of the

tree $(1, \ldots, \log p)$ we obtain the expected complexity of the construction

$$\mathcal{O}\left((t_s + t_w)\log^2 p \log n + (1 + t_w \log p)\frac{nd}{p} + t_s p\right). \tag{9}$$

For large processor counts and large d the communication cost will be excessive due to the $t_w nd/p \log p$ term. More importantly, MPI resources get taxed by managing such a massive exchange (essentially we're shuffling the whole dataset), which leads to having to use very small grain size to avoid running out of memory.

Variant B: One way to resolve this problem is a different point-exchange approach, which we term POINTSPLIT_P2P(): process j with $j < p_\ell/2$ is assigned to c_l otherwise it is assigned to c_r. To redistribute the points, a process with $j < p_\ell/2$ sends its local r_r to $p_\ell - j$ and receives r_l from $p_\ell - j$. This exchange introduces memory/work imbalance (worst case is dn/p), which as we traverse the tree may grow to $\mathcal{O}(nd/p \log p/2)$. In terms of memory, such complexity severely limits scalability. But for median split KD-trees, we can fix this. After the pairwise exchange, we load-balance using an iterative algorithm that uses only point-to-point communication. Let p_ℓ the number of processes that handle some node at level ℓ. Let w_j be the work per process. Then, we sort w_j and use this information to do a pairwise exchange between process j' and the $p_\ell - j'$ process *in the sorted array*. We can show that the maximum number of exchanges required to obtain perfect load balance is $\log p_\ell$ and the overall memory requirement for the construction is $\mathcal{O}(2nd/p)$. The sort can be done using a distributed bitonic sort with time complexity $\log^2 p_\ell$. This scheme removes $\mathcal{O}(t_s p)$ from equation (9):

$$\mathcal{O}\left((t_s + t_w)\log^2 p \log n + (1 + t_w \log p)\frac{nd}{p}\right). \tag{10}$$

Its main value however, is that it avoids collectives and has a lower memory footprint than POINTSPLIT_ALL().

Variant C: Note that neither of the above schemes stores all nodes of the tree at all processes, but only $\log p$ nodes (the path from the root to the leaf). By storing the whole tree, we can remove the $\log p$ factor from the $\mathcal{O}(t_w nd/p \log p)$ point-exchange cost. This

algorithm is described in [3]. The basic structure of the algorithm is exactly the same as Alg 7 but now all the steps are parallelized using all of the processors. Each processor shuffles its points to the appropriate nodes using a top-down approach, with synchronizations at each node. At the leaves, an MPI_ALLTOALLV() redistributes the points to their correct locations. All stages but the final all-to-all are perfectly balanced The drawback is increased costs in computing the median, and the complexity can be simplified to

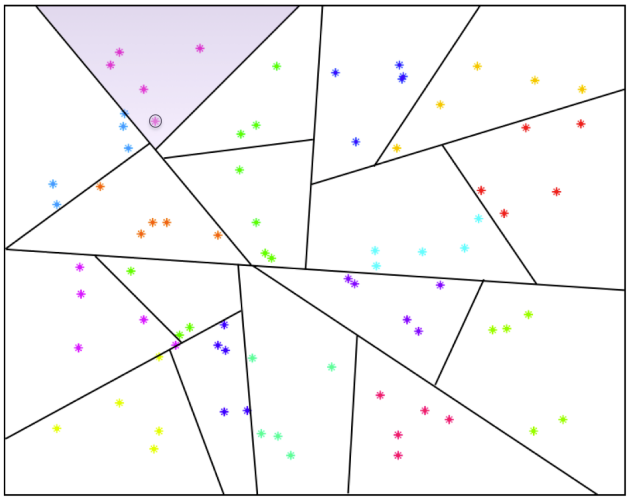
$$\mathcal{O}\left(\log p \log n\,(t_s \log p + t_w p) + (1 + t_w)\frac{dn}{p} + t_s p\right). \tag{11}$$

The last two terms come from the actual point redistribution at the leaves.

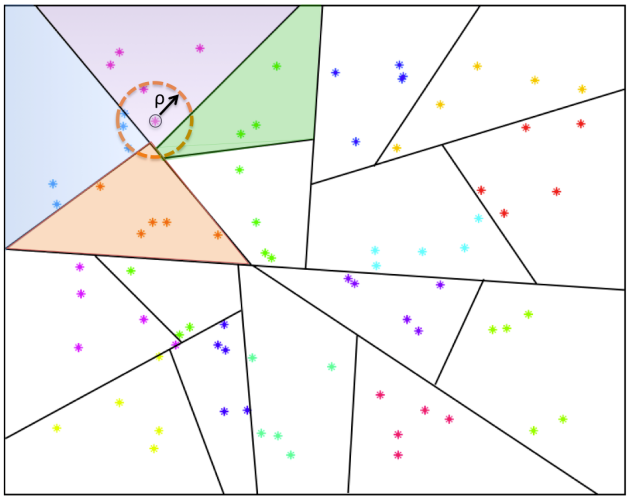
Shared memory. We have discussed the construction of the distributed tree. The shared memory construction is straightforward. Each MPI process owns a leaf of the distributed tree. Its points are used to construct a local tree using Alg 7 and a level-by-level traversal in which we parallelize all the node-splitting and median calculations. The cost is $\mathcal{O}(n/p_t \log(n/s))$, were p_t is the number of threads.

2.3.4.2 Exact Search Algorithm

To exactly search the near neighbor of a query point q, we need to allow q visit one or more leaf nodes in a tree.



(a) greedy tree traversal



(b) range query tree traversal

Figure 9: Illustration of the greedy and exact tree traversal strategies. In a greedy traversal strategy, a query point (marked) as circle only visit the leaf node it belongs to, as in figure (a) the shaded region. In a range traversal, a query point would visit all the leaf nodes which are overlapped with the query's bounding box given by radius ρ.

Range-Query algorithm. This query algorithm is straightforward. Algorithm 10 describes an ρ-near neighbor query on a RKD TREE index for a single query point q. The query begins at the root of the tree. If q's bounding ball $\mathcal{B}_{q,\rho}$ intersects with the space of node i, q will be distributed to the processes which belongs to $\mathcal{C}_i$. Mathematically, that is

$$q \to C_l, \;\; p_{\mathcal{T}} \cdot q - m_{\mathcal{T}} \leq \rho \tag{12}$$

$$q \to C_r, \;\; m_{\mathcal{T}} - p_{\mathcal{T}} \cdot q \leq \rho \tag{13}$$

If points are grouped by clustering, the pruning then becomes

$$q \to C_i, \;\; d(c_i, q) - R_i \leq \rho \tag{14}$$

where c_i is the cluster centroid, and R_i is the cluster radius. As shown in Figure 7, if there are overlaps, a query point shall visit both clusters, which implies it should go to both kid nodes. If the overlap is serious, there can not be good search efficiency.

This process is repeated until the leaf level is reached. At leaf nodes, a direct or LSH based KNN kernel is performed.

Algorithm 10 NODERANGEQUERY(q, ρ, $\mathcal{T}$)

```
1: if ISLEAFNODE(T) then
2:    LEAFRQUERYSEARCH(q, ρ, X_T)
3: else
4:    for each query point q do
5:       if p_T · q − m_T ≤ ρ then
6:          Assign q to C_l
7:       end if
8:       if m_T − p_T · q ≤ ρ then
9:          Assign q to C_r
10:      end if
11:   end for
12:   Gather q to appropriate processes
13:   NODERANGEQUERY(q, ρ, C)
14: end if
```

K-Query algorithm. A k-nearest query can be performed using two traversals of the tree. In the first pass, each query point is assigned to only one of those two leaf node

according to which side of hyperplane it belongs to. Within each of the leaves, an exact KNN search is performed to select an initial search radius for each query point. We call this **greedy** query of a tree, which is described on Algorithm 11. In the second pass, we perform an ρ-query to find *all* neighbors within each query point's search radius. The k nearest of those are then returned.

Algorithm 11 NODEGREEDYQUERY(q, ρ, $\mathcal{T}$)

```
1: if ISLEAFNODE(T) then
2:    LEAFRQUERYSEARCH(q, ρ, X_T)
3: else
4:    for each query point q do
5:       if p_T · q − m_T < 0 then
6:          Assign q to C_l
7:       else
8:          Assign q to C_r
9:       end if
10:   end for
11:   Gather q to appropriate processes
12:   NODEGREEDYQUERY(q, ρ, C_local)
13: end if
```

Our implementation, of course, is for an arbitrary number of query points. This does create a challenge in the case of significant overlap among bounding regions, which may be likely in high-dimensional data sets. In other words, a query points may visit both of the child node when it traverse a tree. In the worst case, each query point may visit all leaf nodes. Then there is no difference between the tree based search and the direct search. To avoid memory exhaustion in these cases, we allow each process to store only a specified maximum number of query points during a tree traversal. If this limit is reached at an internal tree node, we stop the traversal early on that subtree and run an exact or approximate search on that node.

2.3.4.3 Approximate Tree Search Algorithm

The exact tree search algorithm might be in trouble if a query point have to visit many leaf nodes, which is pretty possible in high dimensions, where any points have a similar

distance to each other. Let' s consider the case that two points are both very close to the hyperplane, but locate on the different sides. Suppose there is a third point that is close to these two, it ought to visit both region to find the nearest neigbhors

As a counterpart, we also could visit only one leaf node, as what is described in Algorithm 11. The problem with this approach is it has very low accuracy. Since it is very likely the near neighbors of a query point q are on children nodes if it is near to the hyperplane. However, suppose there is a method which visit the leaf node randomly with an error ϵ, then after r independent runs, the accuracy can be improved to $1 - \epsilon^r$. That is the basic start point of the approximate tree search algorithm.

Our method is based on a sequential algorithm described in [68]. The basic idea is to avoid pruning: rotate the points randomly, build a KD-Tree, and search for nearest neighbors only at the leaves; iterate and combine the results from its iteration till convergence, Alg 12. The points found during the global search are stored in F (say, $F(i)$ contains all the candidate neighbors for point r_i), K stores the solution, and $\mathcal{T}$ is a tree node. (For simplicity, all the algorithms in the section are described for p being a power of two.)

Algorithm 12 RANDOMIZEDKDT_ANN(r)

1: $K = \emptyset, \mathcal{T} =$`root` // initializations
2: $Q =$ RANDOMROTATION
3: $p = Qr$ // with fast rotations
4: NODESPLIT($\mathcal{T}, p$)
5: $F =$LEAFLOCALSEARCH($\mathcal{T}$)
6: $K =$UPDATENEIGHBORS(K,F)
7: `if` CONVERGED(K,F) `return`
8: GOTO 2

Random Projection. As we stated, the primary drawback of standard KD-tree is near points can be partitioned on different sides of a decision plane. As a result, it takes a large fraction of time on back-tracking to prove a candidate point is the true near neighbors. If we only visit one of those two sides, we may obtain low accuracy. However, as illustrated in Figure 10, if we rotate the decision hyperplane a little bit, making those near points on

the same side of the decision hyperplane, only visiting one node might be enough. This inspires the random projection approaches [68].

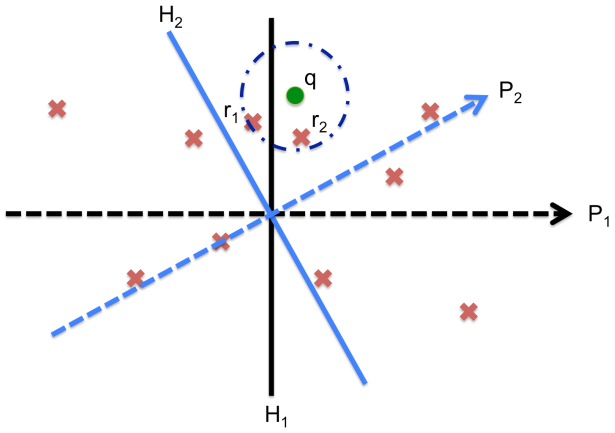


Figure 10: Illustration of random projection tree traversal strategies

Let $\mathcal{T}$ be a tree node, and $\mathcal{T}'$ be the same tree node with the projected points, that is, if the projection matrix is $P_{d\times d}$, then $X_{\mathcal{T}'} = PX_{\mathcal{T}}$, where $X_{\mathcal{T}}$ is the points within this node. The random projection algorithm is then given in Algorithm 13.

Algorithm 13 RANDOMPROJECTIONTREESEARCH(q, $\mathcal{T}_{root}$)

1: **for** i = 1 : T **do**
2: Generate Random Projection Matrix $P^{(i)}$.
3: Rotate Points from $X_{\mathcal{T}}$) to $X_{\mathcal{T}'}$
4: NODESPLIT($\mathcal{T}'$)
5: $\mathcal{N}^i(q)$ = NODEGREEDYQUERY(q, $\mathcal{T}'$)
6: Merge result of $\mathcal{N}(q)$ and $\mathcal{N}^i(q)$ to $\mathcal{N}(q)$
7: **end for**

2.4 Experimental Results

In this section, we present and analyze the performance and scalability results of each of our algorithms on various machines. We examine both single-node kernel performance and scalability to thousands of MPI processes.

Platforms used. Our large strong- and weak-scaling results were obtained from runs on the *Jaguar* system at the National Center for Computational Sciences, *Kraken* platform

at the National Institute for Computational Sciences, and the *Lonestar* cluster at the Texas Advanced Computing Center. Table 3 provides a summary of machine characteristics[4]

Table 3: *Machine characteristics*.

	Jaguar	**Kraken**	**Lonestar**
nodes	18,688	9,408	1,888
cores	299,008	112,896	22,656
GB/*node*	32	16	24
clock (GHz)	2.2	2.6	3.3
GFlops/*core*	8.8	10	13
t_s (μs)	1.9	8.0	2.2
t_w (ns/8 bytes)	0.42	0.43	0.96

All machines have 2 sockets per node and the peak performances are 2.63, 1.17, an 0.3 PFlops/s respectively. Kraken uses AMD's Istanbul architecture connected to a Cray SeaStar2+ router, and the routers are interconnected in a 3-D torus topology. Jaguar uses a Gemini Torus topology. Lonestar is interconnected with QDR InfiniBand in a fat-tree topology and uses Intel Xeon X5680 processors for each socket. The libraries and executables were built using Intel 11.1 with MKL BLAS on Lonestar and the Cray compilers on Jaguar an Kraken with vectorization and OpenMP.

Data set. We test the performance on several datasets, both synthetic and real. The synthetic datasets are points sampled from a normal distribution. Furthermore, we also generate an embedded normal dataset, which we first generate a d-dimensional normal data, then expand them into D-dimensional space by padding zeros and rotation. In this way, the dataset could have an intrinsic dimensionality d. In the following, we denote the embedded normal data as "$md - nD$ emg".

We also use some real datasets. The first one, US Census data, contains a 1% sample of the Public Use Microdata Samples ((PUMS) person records drawn from the full 1990 census sample. It contains about 2.5 millions points with 68 numeric attributes. This

[4]MPI message latency was measured by computing the average time of 1M `MPI_Sendrecv()` calls with a message length of 0. Bandwidth was measured by computing the average time required to transmit a message of of size 1GB. For both measurements, communication occurred between processes on two different network nodes.

dataset is available at the UCI repository [95]. The second real data come from the gabor features of MRI cardiac images; the gabor wavelets is constructed under 3 scales and 8 orientation. The frequency varies according to the scales. Hence, the total dimensionality of gabor features is 96. The third one contains points from the phase space of a chaotic dynamical system, which is of 81 dimension. Roughly speaking, the real life datasets has a low intrinsic dimension. This may imply that the metric tree which are ill defined in high dimensions [129] can be effectively pruned. Figure 11 illustrate statics of some of those datasets.

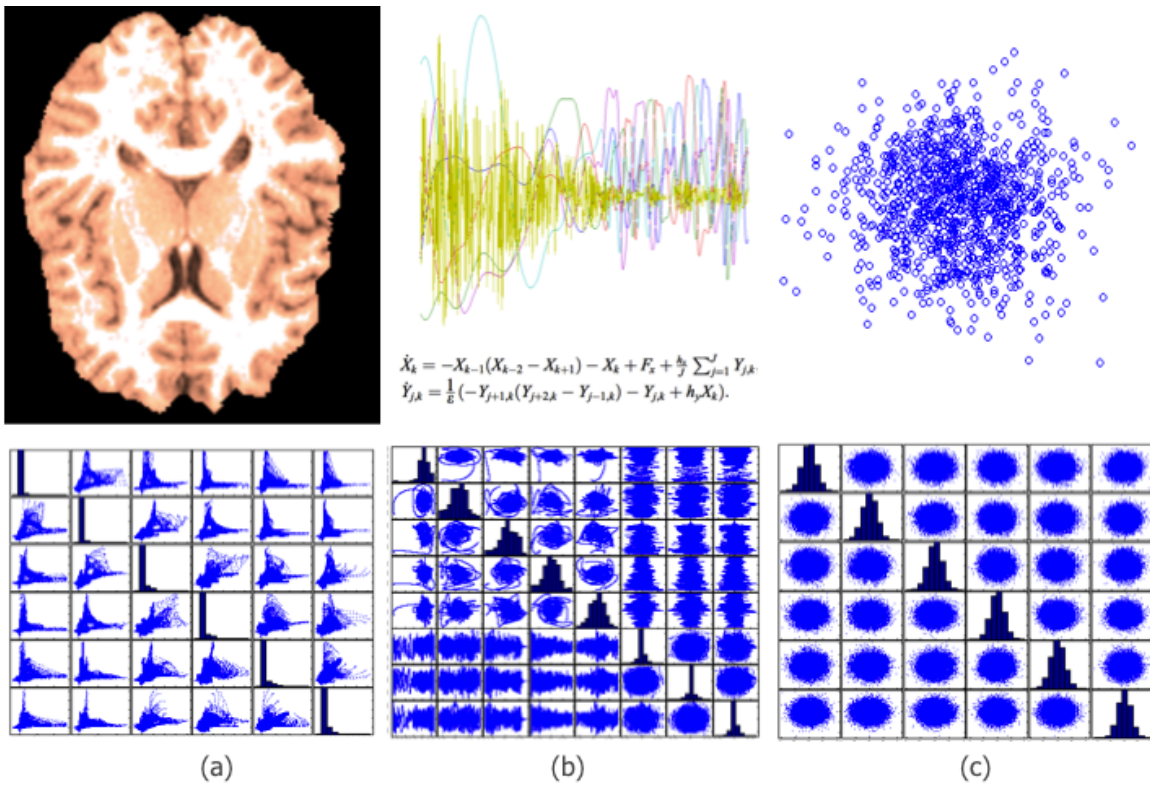


Figure 11: *Data used in the tests:* We use a set of MRI brain images (a), points from the phase space of a chaotic dynamical system (b), and normally distributed points in $\mathbb{R}$ (c). In the bottom row, we show pairwise scatter plots of a six-dimensional slice of the datasets. From these, we can observe the non-Gaussianity of the real-life datasets. Without going into details, this may imply that there is a low intrinsic dimension in the dataset.

Metrics used. When presenting the performance of our various methods, we use the metric *millions of cycles / point / core*, i.e., the number of clock cycles that would be needed to perform the query for each point using only a single core. We use this performance metric

to allow comparisons across systems.

To measure accuracy, we use the *hit rate* and *mean relative error*. Hit rate hit is defined as

$$hit = \frac{1}{nk} \sum_{i=1}^{n} \left| \left\{ \mathcal{N}_i^{true} \right\}_{j=1}^{k} \cap \left\{ \mathcal{N}_i^{found} \right\}_{j=1}^{k} \right| \tag{15}$$

where n is the number of points, k is the number of requested neighbors, $\mathcal{N}_i^{true}$ is the set of true neighbors of point i, and $\mathcal{N}_i^{found}$ is the set of neighbors found by the approximate approach. The mean relative error is the mean of the relative difference between the distances returned from an approximate search and the distances to the exact nearest neighbors. The reason for using both metrics is simple: for a given query point q, an approximate search may not return the exact nearest neighbor, but it may return a point that is almost imperceptibly more distant from q than is the exact nearest neighbor. Because of the prohibitive cost of computing an exact ANN solution for large data sets, we compute both accuracy metrics for a random sample of $\mathcal{O}(\log n)$ points.

2.4.1 Distributed direct query performance.

For the rectangular partitioned direct k-NN, we evaluate performance by conducting strong-scaling tests on 48 to 768 cores (8 to 128 sockets) of Kraken. Our runs were performed with a fixed number of nodes overall, $m = n = 100000$, and with three values of dimension, $d = 8$, 512, and 8192. The results in Table 4 show nearly perfectly linear strong scaling, indicating that the communication and node interdependency caused by the k-reduction does not produce a significant performance bottleneck. In Table 5, we demonstrate perfectly flat weak scaling up to 16384 MPI processes.

For the cyclic direct k-NN, we evaluate performance by conducting weak-scaling tests on 12 to 12,288 cores (1 to 1024 nodes) of Kraken. In this experiment, we use a single MPI process per core and disable multi-threading within processes. Even though we do achieve good performance with hybrid parallelism, this decision was made to enable measuring the scalability to a larger number of independent processes. We test the scalability of both an

Table 4: *Strong scaling of direct k-query with rectangular partitioning* The time required to complete a query and the floating-point performance of the k-NN calculation on Kraken. A square partitioning is used and the number of points is fixed at $m = n = 100000$. One process per socket was used, with 6 OpenMP threads.

p	Time (s)			GFLOP / s		
	$d = 8$	512	8192	8	512	8192
4	1.68	4.33	44.04	0.101	23.7	36.8
8	0.59	1.91	22.09	2.86	53.6	74.2
16	0.37	1.04	11.22	4.57	98.8	146
32	0.15	0.48	5.856	11.7	212	280
64	0.08	0.26	2.932	20.3	400	559
128	0.05	0.14	1.595	33.8	729	1027

Table 5: *Weak scaling of direct k-query with rectangular partitioning* The time required to complete a query and the floating-point performance of the k-NN calculation on Kraken. A square partitioning is used and the number of points per block partition is fixed at $m = n = 5000$. In all runs, $d = 1000$. One process per socket was used, with 6 OpenMP threads.

	$p = 4$	16	4096	16384
Time (s)	27.85	27.83	27.96	27.72
TFLOP / s	0.03	0.11	28.0	111

all-to-all query, and a query set that is significantly larger than the reference set.

The results of the weak scaling tests are summarized in Table 6. The algorithm exhibits nearly perfectly linear weak scaling, which indicates that even for a relatively small problem size per node, the cost of communication is hidden by overlapping it with computation. In fact, for the largest run, the code sustains roughly 40% of peak floating-point performance without any low-level optimization.

2.4.2 Distributed Exact Tree Search Performance.

We have evaluated the performance and scalability of our multilevel tree-based partitioning scheme on the Kraken platform using different data sets.

We firstly compared two grouping strategies, i. e., the hyperplane partition and the clustering grouping. For the exact tree search, the most important evaluation is how many

Table 6: *Weak scaling of cyclic algorithm* The time required to complete a query and the floating-point performance of the k-NN calculation on Kraken. In all runs, $d = 100$. We use one process per core

p	$m = n = 1000$		$m = 100000, n = 100$	
	Time (s)	TFLOP/s	Time (s)	TFLOP/s
96	4.58	0.44	49.1	0.38
192	9.30	0.80	97.7	0.76
384	18.4	1.61	195	1.53
768	36.9	3.23	403	2.96
1536	73.9	6.45	815	5.85
3072	151	12.6	—	—
6144	303	25.2	—	—
12288	604	50.5	—	—

points (or leaf nodes) a query should search to find the exact true neighbors. In the worst case, it should search for its neighbors in all reference points, then there is no difference between the tree structure and the simplest direct search. The ideal case should be each leaf node has the same number of query points. Thus, to measure the efficacy of the tree, we define the pruning as

$$prune_i = \frac{N^q - n_i^q}{N^q - N^q/p} \tag{16}$$

where N^q is the total number of query points among all processes; n_p^q is the number of query points on a single process i; p is the number of processes.

Table 7 shows the pruning percentage of both hyperplane partition and clustering grouping on 5 different data sets. Generally speaking, all those 5 data sets has lower intrinsic dimensionality than their apparent dimensions. As a result, even the dimension is high, there are some pruning that could be obtained. The prunability is crucial in two aspects. Firstly, if the data set cannot prune well, query points should visit most of the leaf nodes to find the true neighbors, which loses its advantage over the direct search. Secondly, as we can see, the bad pruning will introduce load imbalance of query points, which means some leaf nodes has more points, the others has less. Consequently, some processes have more work than others, we can not fully make use of the machine. On the other hand, we could find

Table 7: *Pruning Effect of* RKD TREE *on different data set.* Tree is constructed using two grouping method, i. e., hyperplane partition and clustering grouping. The pruning is obtained using 5 different data sets. The US census run uses 9,100 reference points and 500 query points per process, and totally 256 processes. All the other 4 runs use 10,000 reference points and 500 query points per process, and totally 1,024 processes. The maximum, minimum and average pruning percentage across all processes are reported.

dataset	hyperplane partition (100%)			clustering grouping (100%)		
	min	max	**avg**	min	max	**avg**
US census	45.90	99.98	**89.92**	32.87	99.87	**63.83**
dynamical system	97.76	99.97	**98.99**	0.6639	93.49	**15.39**
gabor wavelets	96.85	100.00	**99.01**	54.08	100.09	**88.24**
5d-100D emg	99.66	99.84	**99.75**	4.17	97.95	**57.74**
10d-100D emg	83.96	94.32	**89.82**	0	3.38	**0.5456**

the clustering grouping has less pruning than the hyperplane partition on all the 5 datasets. Hence, the load imbalance would be serious in the clustering grouping method. Since the hyperplane partition could promise a perfect load balance in the reference points if we split the points using the median of the values that points projected along the projection direction. As a counterpart, it is very difficult to cluster points into clusters which have exact the same size. As we split a node, we will assign one or more clusters for each of its kids, there is no guarantee then each kid has the same number of reference points. Hence, we prefer the hyperplane partition strategy, and in the following scaling test, we only present the hyperplane partition approach.

Table 8 gives the pruning effect with different k for RKD TREE using hyperplane partition. As k increases, the search radius for each query point would increase. Thus, it is possible to look at more points to find the nearest neighbors. On the other side, the decrease of pruning affected by the k is not as serious as d, the intrinsic dimensionality. For example, for the $5d - 100D$ embedded normal data set, as k goes from 2 to 50, the average pruning only decrease 0.6%; however, when we increase the intrinsic dimensionality to 10, the average pruning decrease about roughly 10%. This is also a demonstration that the key issue with tree structure for exact search is the intrinsic dimensionality.

Table 8: *Pruning Effect of* RKD TREE *using hyperplane partition with different* k. Tree is constructed using hyperplane partition. The pruning is obtained using 3 different data sets. The US census run uses 9,100 reference points and 500 query points per process, and totally 256 processes. All the other 2 runs use 10,000 reference points and 500 query points per process, and totally 1,024 processes. The maximum, minimum and average pruning percentage across all processes are reported.

k	US census			gabor wavelets			5d-100D emg		
	min	max	avg	min	max	avg	min	max	avg
2	45.90	99.98	89.92	96.85	100.00	99.01	99.66	99.84	99.75
10	36.81	99.86	86.34	96.11	99.99	98.54	99.73	99.40	99.54
20	33.56	99.84	84.50	95.67	99.98	98.23	99.2	99.62	99.40
30	31.90	99.83	83.32	95.10	99.97	97.98	98.98	99.59	99.29
40	30.79	99.82	82.43	94.74	99.96	97.82	98.87	99.54	99.20
50	29.97	99.81	81.70	99.96	94.52	97.71	98.74	99.50	99.11

Table 9: *Weak scaling of* RKD TREE k*-query (*$k = 2$*) on the* embedded normal *data set.* The time (in seconds) required for tree construction and query on Kraken with a fixed number of points per process. 10000 reference points and 1000 query points per process were used for the 100-dimensional runs (points are sampled from a 5-dimensional normal distribution, and embedded into 100-dimensional space.) We use one MPI process per core.

	$p =$	192	384	768	1536	3072	6144	12288
$5d$	const. (s)	0.81	0.99	1.27	1.46	1.69	2.44	2.80
	query (s)	3.82	4.25	5.10	5.81	13.07	18.92	28.42
	% prune	98.86	99.40	99.68	99.83	99.91	99.95	99.97
$10d$	const. (s)	0.93	0.96	1.26	1.83	2.07	3.99	3.14
	query (s)	37.61	55.92	81.57	115.72	162.11	267.27	379.75
	% prune	75.04	82.41	87.99	91.96	94.74	96.54	97.56

We evaluate the weak scaling performance of the tree search on 128 to 24,576 cores (64 to 2048 nodes) of Kraken. In this experiment, we use a single MPI process per core and disable multi-threading within processes. Even through we do achieve good performance with hybrid parallelism, this decision was made to enable measuring the scalability to a larger number of independent processes. Performance is summarized in Table 9. We see that the construction time required to partition and redistribute the points grows very slowly as a function of problem size and process count. However, for the query points, there is some overlap in the bounding regions of the clusters, some number of query points

will be replicated to multiple kids in the tree traversal. Since we test the weak scaling using data sampled from a normal distribution, as the process count increase, the density of data points becomes more and more large. Thus, it is likely that at the very beginning, there is a big overlap near the hyperplane. As going to deeper level, the overlap might be reduced. The query time increase quickly as the process count increases. This is mainly because as the process count increases, although the pruning percentage also is improved, the number of query points becomes larger on each leaf node, which means we should perform the direct nearest neighbor search kernel in a larges scale of data points. For example, when $p = 12288$, the maximum number of query points on a single leaf node is 417065, compared with 65951 for $p = 192$, which is 6.3 times larger. And to redistribute large amount of points also takes longer time. Generally speaking, the pruning degrades since the point density of the generated data set increases rapidly as the problem size is increased.

2.4.3 Distributed `LSH` Performance

Overall, our distributed `LSH` k-nearest query algorithm exhibits quite good performance and scalability.

The strong-scaling performance of the `LSH` algorithm is shown in Figure 12. Each of the queries were auto-tuned and run with the values of $\mathcal{V}$ and $\mathcal{L}$ determined to be optimal; therefore, these parameters do vary with p. In all cases, the auto-tuner was run with a target bucket size of 200 and correctness probability 0.5. The total number of buckets is fixed at 32,000 to ensure a relatively constant amount of work in all runs. Overall, the algorithm exhibits the expected speedup up to 512 sockets (3072 cores) of Lonestar. While the construction phase does not continue to scale past 64 processes, this is to be expected since decreasing the amount of evaluation-phase work each process performs necessitates increasing the value of $\mathcal{V}$ to maintain a good load balance. This, in turn, results in a higher value of $\mathcal{L}$ and increased communication cost.

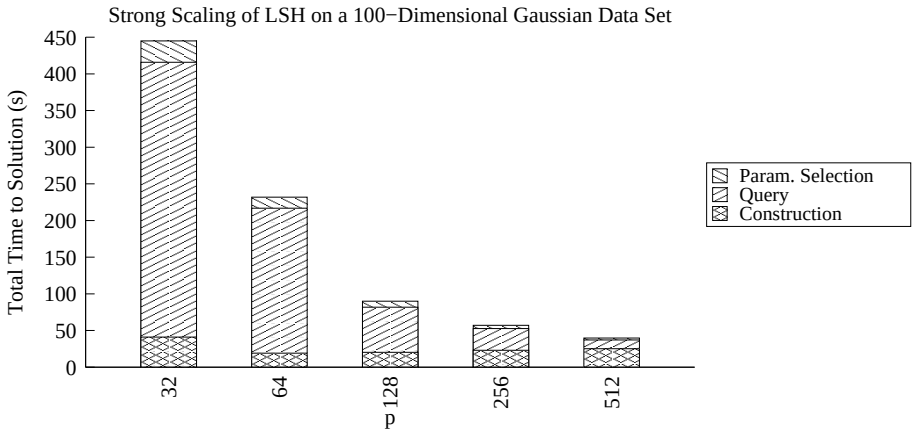


Figure 12: *Strong-scaling of distributed* `LSH` *on a 100-dimensional Gaussian data set on Lonestar:* This graph shows the scalability of the various phases of the distributed `LSH` query. In these runs, an all-to-all 2-nearest query was performed on a data set consisting of 6.4M points. 1 MPI process per socket was used with 6 OpenMP threads. The queries were auto-tuned with a fixed target bucket size of 200. To ensure that the work in each run is relatively constant, the *total* number of buckets is fixed at 32,000.

Strong-scaling results on Kraken are presented in Table 10 for both the embedded and Gaussian datasets in 96 dimensions. The query phase exhibits good scalability up 4,096 processes. However, the constuction phase does not continue to scale past 2,048 sockets because of the increasing weight of the `MPI_Alltoallv()` latency cost relative to computational work. The higher query-phase cost of the embedded dataset vs. the Gaussian is the result of the auto-tuner choosing a significantly higher value of $\mathcal{V}$ for the Gaussian data points. This has the effect of greatly reducing the number of distance calculations that occur in each iteration at the cost of a significantly slower convergency rate. After 8 iterations, the hit rate for the embedded dataset is at least 89% (depending on p); whereas it is less than 51% for the Gaussian.

Table 11 demonstrates the the weak scaling of the algorithm on various data sets with a fixed $\mathcal{V} = 32$ and $\mathcal{L} = 50$. The search radius is still chosen to be appropriate for each run. We use one MPI process with 6 OpenMP threads per socket and problem size of 100,000 points per process. The table shows (from left to right) the table construction time, the time required to query the tables, the maximum time a process spends in direct

Table 10: *Strong Scaling of* `LSH` *on Kraken:* Strong scaling of 8 iterations of `LSH` on Kraken for the a $96d$ gaussian data set and a $10d$ gaussian embedded in 96 dimensions. The total per-iteration time for the smallest embedded run was 121.1s; the time for the largest was 6.9s (excluding tuning).

cores		512	1024	2048	4096
Gaussian	param.	44.06	19.89	9.88	4.99
	const.	19.50	9.40	6.08	32.32
	query	244.24	74.00	31.43	21.49
embedded Gaussian	param.	37.95	20.79	8.92	4.71
	const.	17.66	9.50	6.04	28.14
	query	950.97	293.46	52.79	27.68

evaluations, the minimum time a process spends in direct evaluations, the overall search time (exclusive of ρ selection), and the estimated time needed to perform a direct search using our distributed cyclic algorithm on the same number of processors. The results show that the algorithm generally scales well on a large cluster with a commodity interconnect. Although the performance does degrade for the larger runs in higher dimensions, this is to be expected since the fixed value of $\mathcal{V}$ becomes insufficient for the larger problem sizes, resulting in a communication imbalance in both the table construction and result collection. This behavior does not occur when $\mathcal{V}$ is auto-tuned; however, the choice was made to use fixed values for $\mathcal{V}$ and $\mathcal{L}$ in this table to allow the direct comparison of the algorithm's performance for differing data distributions and spatial dimensions. In this table, we also estimate the time needed to run a direct search on each data set. In the case of the 100-dimensional data sets, a direct search is the only viable method of computing an exact solution.

Figure 13 demonstrates both the performance and necessity of our parameter auto-tuning algorithm. While auto-tuning $\mathcal{V}$ does increase the cost of the parameter-selection phase by roughly a factor of 4-5, the significant improvement in construction time, query time, and overall scalability more than offsets this cost. In fact, with auto-tuning enabled, the overall execution time of the 256-process run on the 4-dimensional data set is only

Table 11: *Weak scaling of distributed* `LSH` *on Lonestar with fixed $\mathcal{V}$ and $\mathcal{L}$.*

Distrib.	Dim.	Proc.	Const.	Query	Eval. Max	Eval. Min	Overall	Direct Search Time
Uniform	4	32	3.69	3.42	2.05	1.91	7.13	1,996
		64	4.60	4.04	2.23	1.92	8.67	3,999
		128	7.70	4.83	2.34	1.91	12.58	7,983
		256	6.71	5.89	2.25	1.90	12.66	16,159
		512	10.56	3.88	2.20	1.92	14.45	32,504
	10	32	5.04	3.13	1.81	1.68	8.20	2,015
		64	5.96	3.46	1.88	1.68	9.46	3,993
		128	8.75	4.22	2.01	1.68	13.03	8,033
		256	16.64	6.77	2.80	1.71	23.46	16,183
		512	19.55	4.79	2.54	1.71	24.38	32,457
	100	32	24.51	14.53	9.39	7.65	39.11	2,844
		64	25.90	19.29	12.31	10.49	45.28	5,705
		128	30.59	29.32	19.80	17.07	60.05	11,976
		256	44.76	32.34	24.01	21.65	77.25	23,020
		512	49.63	41.07	30.05	23.98	90.73	46,382
Gaussian	4	32	3.72	3.53	2.09	1.91	7.28	1,996
		64	4.81	3.77	2.21	1.92	8.63	3,999
		128	7.72	4.83	2.32	1.92	12.60	7,983
		256	6.74	5.98	2.24	1.91	12.79	16,159
		512	10.22	3.70	2.18	1.91	13.93	32,504
	10	32	5.06	3.27	1.86	1.68	8.37	2,015
		64	5.98	3.48	1.92	1.69	9.50	3,993
		128	8.95	3.91	2.02	1.68	12.92	8,033
		256	16.90	6.93	2.82	1.79	23.87	16,183
		512	19.72	4.53	2.30	1.73	24.30	32,457
	100	32	26.19	14.47	9.22	7.97	40.73	2,844
		64	29.90	21.51	13.37	11.10	51.54	5,705
		128	34.35	31.08	20.14	17.06	65.55	11,976
		256	39.80	36.85	25.08	22.12	76.78	23,020
		512	49.09	50.21	37.31	27.95	99.45	46,382
Mixture of Gaussians	4	32	3.97	3.37	2.06	1.91	7.37	1,996
		64	5.10	3.75	2.17	1.90	8.90	3,999
		128	8.16	4.85	2.38	1.92	13.06	7,983
		256	7.26	6.01	2.24	1.92	13.32	16,159
		512	10.44	3.72	2.22	1.92	14.16	32,504
	10	32	5.08	3.10	1.77	1.67	8.21	2,015
		64	6.13	3.48	1.91	1.67	9.65	3,993
		128	9.08	4.18	2.05	1.68	13.3	8,033
		256	18.26	6.98	2.85	1.85	25.29	16,183
		512	17.85	5.76	2.53	1.92	23.66	32,457
	100	32	25.57	12.41	7.87	6.71	38.05	2,844
		64	28.64	18.01	11.79	10.04	46.76	5,705
		128	32.20	20.91	12.93	11.26	53.25	11,976
		256	55.25	52.75	39.91	29.87	108.20	23,020
		512	61.80	72.27	52.12	34.28	134.18	46,382

40% greater than that of the 32-process run, including parameter selection. Contrasted to the 218% increase in runtime without auto tuning, this is a very noticeable improvement. Note also that the mean distance error for the largest auto-tuned run is only 1.1% using a success probability of 0.5. Recall from table 11 that a direct search on the same problem would require over 16000s, a difference of over 3 orders of magnitude (including parameter selection time). While the difference is less dramatic for the 100-dimensional data set, auto-tuning $\mathcal{V}$ still results in significantly improved scalability. It should also be stressed that the diminished effect of auto-tuning in the 100-d case is simply because the non-tuned $\mathcal{V}$ selected is coincidentally close to the optimal value.

While the performance of `LSH` is dataset-dependent, we have demonstrated the scalability of the overall method up to 240K cores of Jaguar, reaching 20% of peak floating-point performance. Because of the communication-heavy nature of the algorithm, this is a significant result. The weak-scaling results of `LSH`-ANN searches on Jaguar are presented in Figure 14.

Figure 14 (a) is the case that the query and reference points are different. 50,000 total points per process were used, divided evenly between the $\mathcal{R}$ and $\mathcal{Q}$ sets. The per-iteration execution time (excluding tuning) for the smallest run was 19.6s; the largest run took 47.3s. The primary cause for the lower efficiency when compared to the $\mathcal{Q} = \mathcal{R}$ case is the poor scaling of MPI_Alltoallv with large message sizes. Since the $\mathcal{Q} \neq \mathcal{R}$ involves twice the number of collective calls, the latency costs and communication imbalance are amplified. Obviously, this presents an opportunity for further optimization. Figure 14 is the all-to-all nearest neighbors search. 50,000 points per process were used, with the entire data set used as both $\mathcal{R}$ and $\mathcal{Q}$. The per-iteration execution time (excluding tuning) for the smallest run was 105.7s; the largest run took 124.1s, 24.3s of which was spent in MPI collectives to exchange points and results. In the large runs, a greater than expected imbalance (roughly 75%) was observed. Further optimizations to our direct kernel have improved the balance, and a repeat of our 9k-process run shows that the imbalance is reduced to 50%. This is the

best balance that can be guaranteed because of the large-bucket splitting. The largest mean relative distance error was 1.47%; the lowest hit rate was 88.3%.

For all LSH weak-scaling runs, we set the total number of hash buckets to be $10p$. This value is slightly smaller than optimal for the $\mathcal{Q} = \mathcal{R}$ runs and results in a moderate load imbalance; however the performance remains quite good. The results show that we are able to maintain a high level of efficiency despite the significant communication cost. Weak-scalability results for $\mathcal{Q} \neq \mathcal{R}$ queries up to 72K cores are presented in Figure 14. While the added communication cost inherent to this type of query results in lower efficiency, we still maintain good scalability.

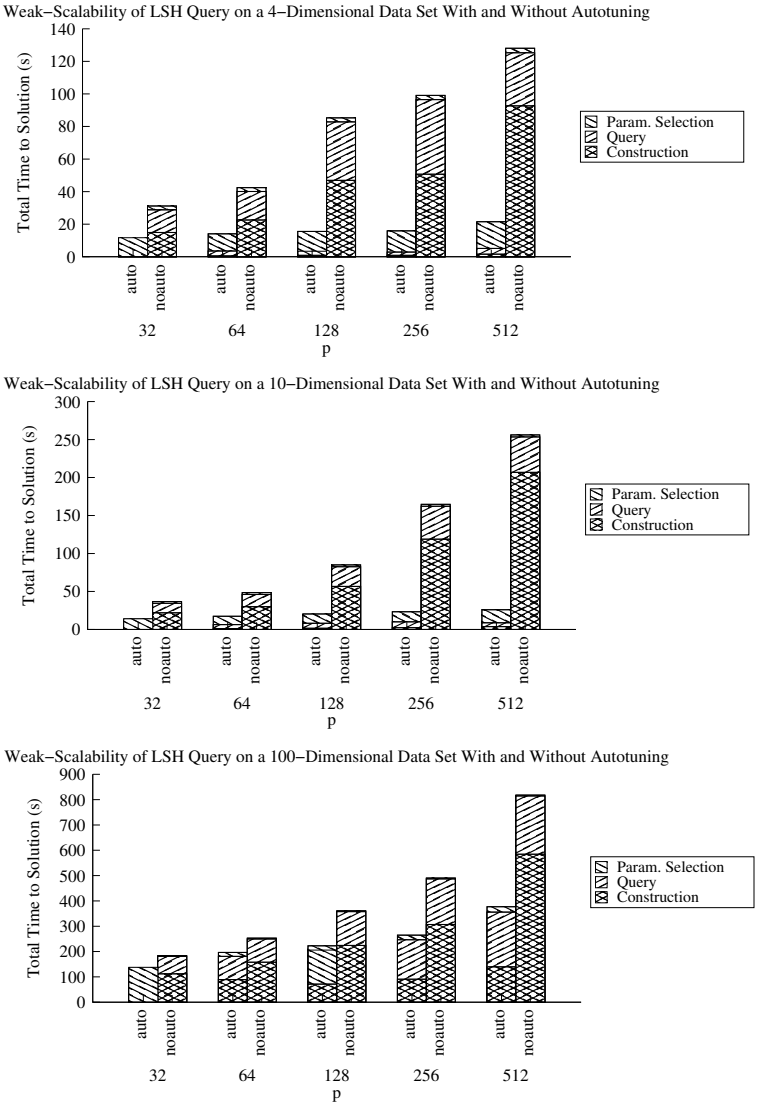
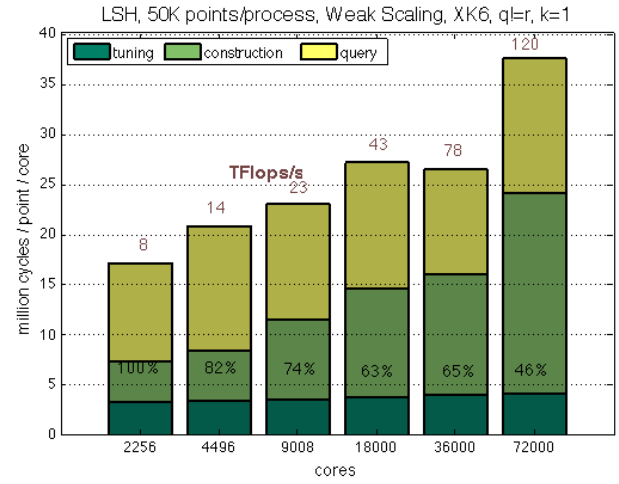
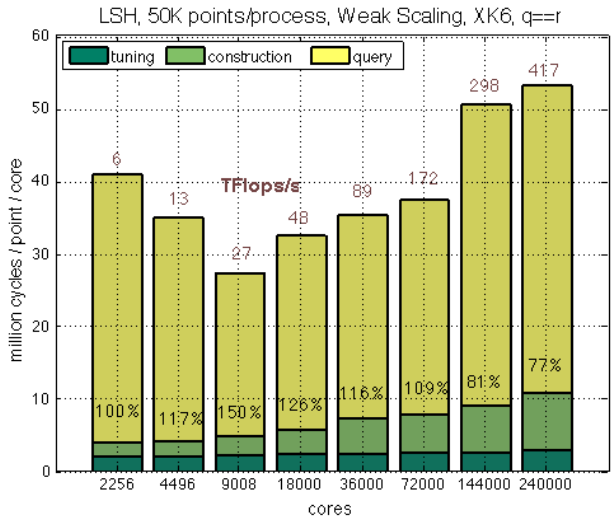


Figure 13: *Weak scaling of distributed* `LSH` *both with and without auto-tuning on Lonestar:* This graph demonstrates the importance of auto-tuning the parameter $\mathcal{V}$ for each data set. Note that each graph has a different y-axis scale. The "Param. Selection" time includes both the selection of ρ and (when enabled) the auto-tuning of $\mathcal{V}$. The auto-tuned results were obtained with a target bucket size of 200 and correctness probability $P = 0.5$. A problem size of 100,000 points per process was used, with one MPI process and 6 OpenMP threads per socket. The mean relative distance error for the auto-tuned runs was roughly 1%-3%, depending on d. For the non-auto-tuned runs, we use $\mathcal{V} = \mathcal{V}_0$ (see section 2.3.3.1).



(a) $\mathcal{Q} \neq \mathcal{R}$



(b) $\mathcal{Q} = \mathcal{R}$

Figure 14: *Embeded Normal Dataset:* Weak scaling of 8 iterations of `LSH` on Jaguar for a $10d$ gaussian data set embedded in 2048 dimensions with $k = 1$. We run one MPI process per NUMA node with 8 OpenMP threads per process. We use a total of $10p$ hash buckets in each run. The x-axis is the total number of cores/threads. The percentages given represent the relative efficiency of each run normalized to the 2,256-core run.

2.4.4 Distributed Approximation Tree Search Scalability.

In our first experiments, we studied the scalability of the tree construction algorithms on Lonestar. We compare the three tree construction schemes on that system in Table 12. The lower bandwidth of Lonestar's interconnect combined with its less-tuned MPI stack clearly shows the differences in the scalability of the three methods. Weak-scaling results for Variants B and C on Kraken are presented in Table 13. Here too, Variant C performs well at small p but does not scale well; however, Variant B maintains very good efficiency up to 96K cores. Table 14 presents weak-scaling results for all three tree variants on Jaguar. On Jaguar, the differences between the three construction algorithms is less marked, with the exception of the fact that Variant C also fails to scale to large p on this machine. It is not currently clear why Variant B fails to maintain the same level of parallel efficiency on Jaguar that it exhibits on Kraken at comparable core counts. Further investigation is necessary to explain this behavior.

Table 12: *Weak-scaling of Tree Construction on Lonestar.* This table shows the scalability of the three tree construction variants on Lonestar in terms of millions of cycles per point per core and the efficiency relative to the 192-core run. We use one process per socket with 6 OpenMP threads. We use 10K points per process in 2,048 dimensions generated using the embedded data generator. On this system, Variant C performs well at small scale, but Variant B shows the best overall scalability. For reference, the tree construction times for each iteration of the smallest runs were 4.0s, 2.6s, and 2.15s for Variants A, B, and C, respectively. The times for the largest runs were 33.1s, 5.7s, and 6.6s.

Tree: Lonestar weak scaling						
cores	192	384	768	1,536	3,072	6,144
VARIANT A						
cycles	10.7	17.8	25.3	39.1	58.0	88.5
effic	100%	60%	42%	27%	18%	12%
VARIANT B						
cycles	6.9	8.4	9.8	11.3	13.0	15.2
effic	100%	83%	71%	62%	54%	46%
VARIANT C						
cycles	5.7	7.3	8.4	10.2	12.6	17.6
effic	100%	78%	69%	56%	45%	32%

Table 13: *Weak-scaling of Tree Construction on Kraken*. This table shows the scalability of tree construction variants B and C on Kraken in terms of millions of cycles per point per core and the efficiency relative to the 1.5K core run. We use 10K points per process in 2,048 dimensions. Variant B demonstrates a clear scalability advantage. The construction times per iteration for the smallest runs were 8.63s and 6.73s for Variants B and C, respectively. The times for the 24K-core runs were 14.4s and 28.5s. The construction time for Variant B at 96K cores was 20.7s.

Tree: Kraken weak scaling							
cores	1536	3072	6K	12K	24K	48K	96K
VARIANT B							
cycles	18.0	17.9	21.2	25.8	30.1	36.0	43.1
effic	100%	100%	85%	70%	60%	50%	42%
VARIANT C							
cycles	14.0	12.5	20.7	31.7	59.3	-	-
effic	100%	112%	68%	44%	24%	-	-

The strong scaling performance of random projection tree is illustrated in Figure 15 for both the embedded and Gaussian datasets in 96 dimensions. The query phase does not continue to scale past 2,048 processes, and the construction phase does not exhibit good scalability. That is because the weight of the `MPI_Alltoallv()` latency cost relative to computational work increases as the number of processes goes up. However, compared to `LSH`, it can be seen that the tree structure gives a faster the search, and the accuracy does not vary as the number of process changes (always 95% for the embedded dataset and 51% for the gaussian dataset). Another difference is the query cost for both datasets are similar, due to the hyperplane partition strategy, unlike `LSH`, the tree algorithm do not need to tune parameters for different distributions of data, although the distribution do play an important role on the accuracy for the approximation search. In other words, the tree algorithm is more stable for data from different distributions, since the construction scheme does not depend on the structure of data itself.

Finally, we illustrate the weak scaling performance of the whole random projection tree search algorithm up to 130K cores on Jaguar. The weak scaling experiments run on the 2,048 dimensional embedded dataset. We tested all three variants of tree structures for

Table 14: *Weak-scaling of Tree Construction on Jaguar*. This table shows the scalability of the three tree construction variants on Jaguar in terms of millions of cycles per point per core and the efficiency relative to the 2048-core run. We use one process per socket with 8 OpenMP threads. We use 10K points per process in 2,048 dimensions generated using the embedded data generator. On Jaguar, the difference between the three variants is less noticeable at small scale. However, here again, Variant C provides the best performance at small p but does not scale well at large p. The reason for the large spike at 64K cores is unclear, however. Variant B exhibits slightly better performance and scalability than Variant A. The construction times for each iteration of the smallest runs were 6.3s, 5.6s, and 3.02s for Variants A, B, and C, respectively. The construction times for the 64K-core runs were 26.2s, 22.1s, and 75.2s. The time for the 128K-core run was 29.1s for A and 28.3s for B.

Tree: Jaguar weak scaling							
cores	2048	4K	8K	16K	32K	64K	128K
VARIANT A							
cycles	11.1	16.4	20.3	27.3	33.5	46.1	51.1
effic	100%	68%	55%	41%	33%	27%	22%
VARIANT B							
cycles	9.8	12.5	15.8	22.9	32.4	38.8	49.8
effic	100%	78%	62%	43%	30%	25%	20%
VARIANT C							
cycles	5.3	8.1	10.7	25.5	34.2	132.3	-
effic	100%	66%	50%	21%	16%	4%	-

Table 15: *Strong Scaling of* RKD TREE *on Kraken:* Strong scaling of 8 iterations of RKD TREE on Kraken for the a $96d$ gaussian data set and a $10d$ gaussian embedded in 96 dimensions. 'const.' stands for tree construction time and 'query' is the querying time. The total per-iteration time for the smallest embedded data was 20.8s; the time for the largest was 2.9s. The gaussian distributed data has a similar running time.

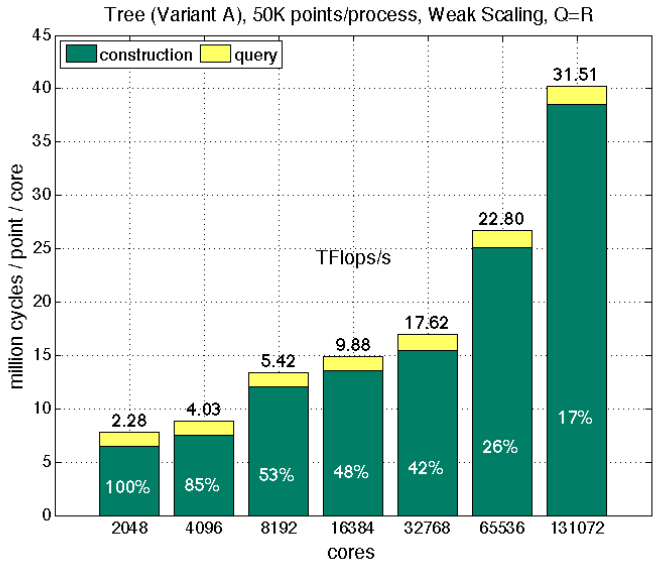
cores		512	1024	2048	4096	8192
Gaussian	const.	139.80	85.56	47.83	27.06	19.54
	query	26.67	12.93	6.03	4.12	3.93
embedded Gaussian	const.	142.44	81.59	48.60	26.70	19.47
	query	25.34	12.04	6.31	4.16	3.89

both $\mathcal{Q} = \mathcal{R}$ and $\mathcal{Q} \neq \mathcal{R}$ cases, and the results are presented in Figure 15 (Variant A), 16 (Variant B) and 17 (Variant C) respectively. It is clear that the construction part is the most expensive portion due to the comprehensive data split and exchange across processes

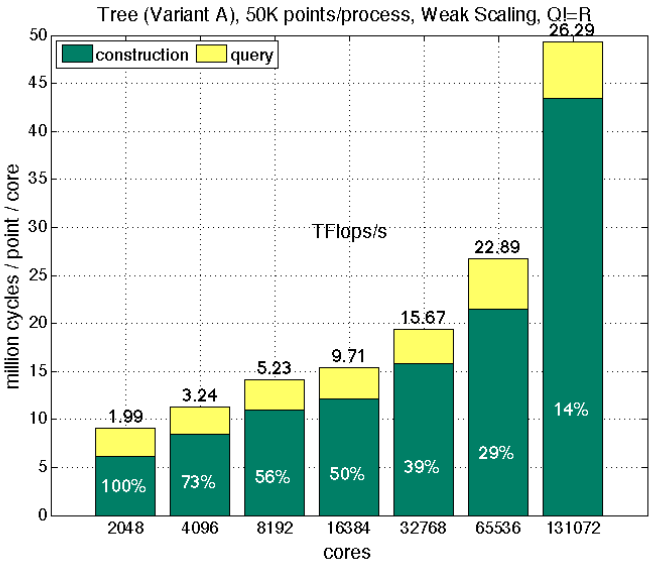
at every level. However, after the tree is built, querying is very easy. Each query point only need to go through the tree to the most possible node, and search for the neighbors only within this node. This implies the tree algorithm is very suitable for some kind of applications where it is possible to pre-build the trees. Besides, compared with `LSH`, the tree algorithm is faster.

Discussion. We used the normalized MCycles statistic to compare performances. A smaller number is obviously better. For weak scaling, increasing numbers indicate communication overheads and load imbalance. Notice that algorithmically we show that the best-case complexity estimates include terms that grow with powers of $\log p$, linearly in p, or with factors that depend on $\log n$ that inherently limit the attainable efficiencies.

This is the first time that ANN solvers have been scaled to this extent and a first attempt to characterize the parallel scalability of state-of-the-art approximate ANN methods in leadership architectures and as such it is incomplete. However, there are some general conclusions that can be drawn. From Figure **??**, we see that LSH requires roughly 50 MCycles/point/core, very similar to the numbers for the two of the tree variants in Table 14. The clear loser in most of the cases is Variant C, but again there are regimes in which it performs well (small number of core counts). The interplay between latency, bandwidth, bucket (or points per leaf node) size suggests significant opportunities for performance tuning and optimization depending on the machine architecture and the dataset and its dimensionality.

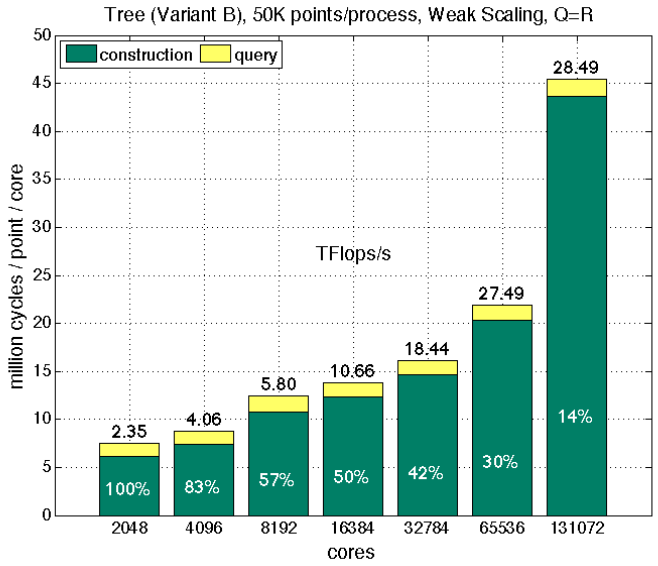


$\mathcal{Q} = \mathcal{R}$

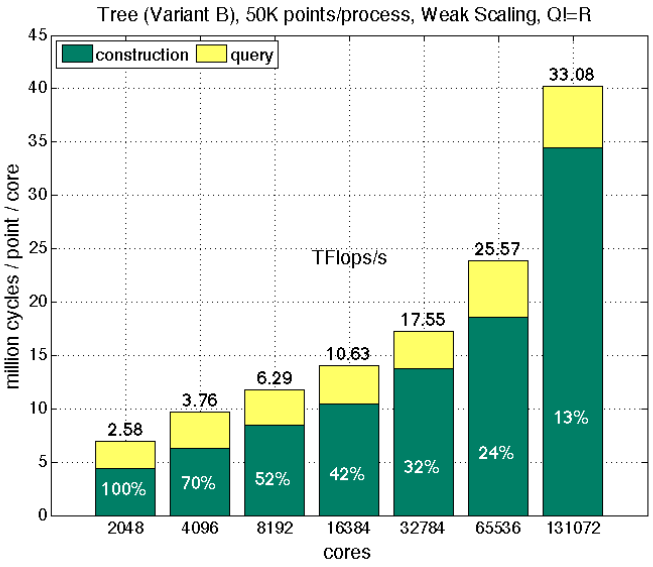


$\mathcal{Q} \neq \mathcal{R}$

Figure 15: *Embeded Normal Dataset with Tree Variante A:* Weak scaling of 8 iterations of RKD TREE in `Variant A` on Jaguar for a $10d$ gaussian data set embedded in 2048 dimensions with $k = 1$. 50,000 points per process were used, with the entire data set used as both $\mathcal{R}$ and $\mathcal{Q}$. We run one MPI process per NUMA node with 8 OpenMP threads per process. We use a total of $10p$ hash buckets in each run. The x-axis is the total number of cores/threads. The percentages given represent the relative efficiency of each run normalized to the 2,048-core run.

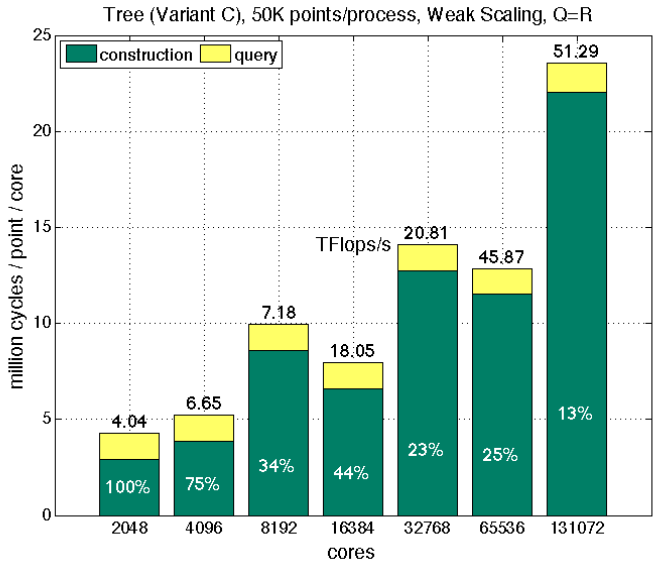


$\mathcal{Q} = \mathcal{R}$

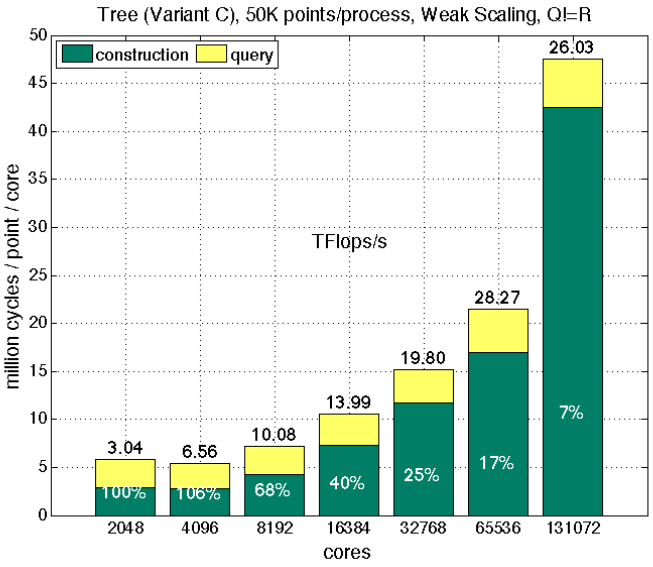


$\mathcal{Q} \neq \mathcal{R}$

Figure 16: *Embeded Normal Dataset with Tree Variante B:* Weak scaling of 8 iterations of RKD TREE in `Variant B` on Jaguar for a $10d$ gaussian data set embedded in 2048 dimensions with $k = 1$. 50,000 points per process were used, with the entire data set used as both $\mathcal{R}$ and $\mathcal{Q}$. We run one MPI process per NUMA node with 8 OpenMP threads per process. We use a total of $10p$ hash buckets in each run. The x-axis is the total number of cores/threads. The percentages given represent the relative efficiency of each run normalized to the 2,048-core run.



$\mathcal{Q} = \mathcal{R}$



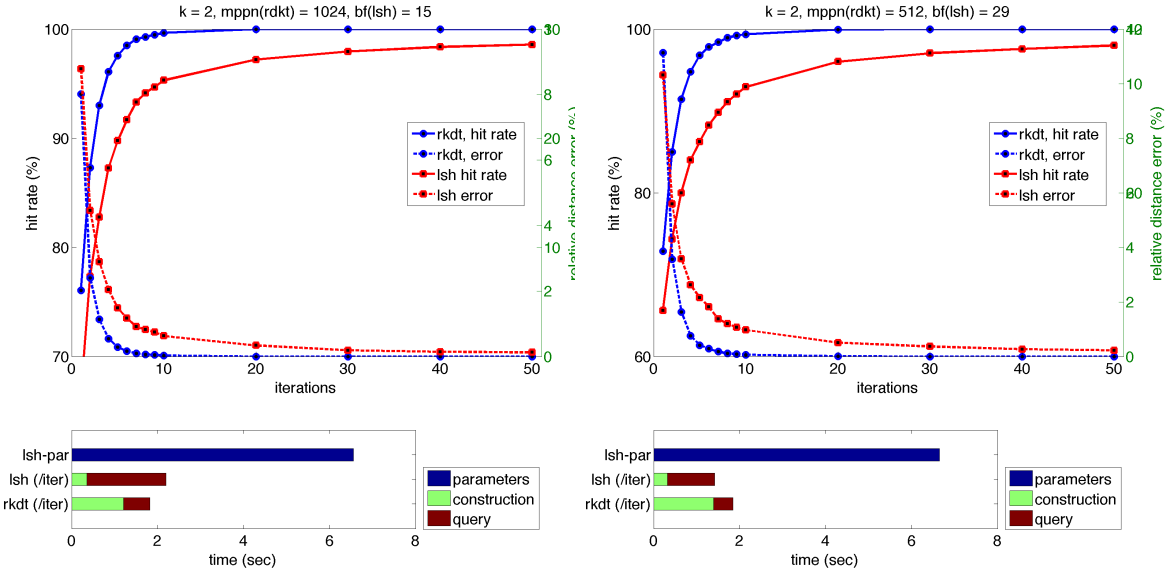
$\mathcal{Q} \neq \mathcal{R}$

Figure 17: *Embeded Normal Dataset with Tree Variante C:* Weak scaling of 8 iterations of RKD TREE in `Variant C` on Jaguar for a $10d$ gaussian data set embedded in 2,048 dimensions with $k = 1$. 50,000 points per process were used, with the entire data set used as both $\mathcal{R}$ and $\mathcal{Q}$. We run one MPI process per NUMA node with 8 OpenMP threads per process. We use a total of $10p$ hash buckets in each run. The x-axis is the total number of cores/threads. The percentages given represent the relative efficiency of each run normalized to the 2,048-core run.

2.4.5 Comparison between LSH and RKD TREE

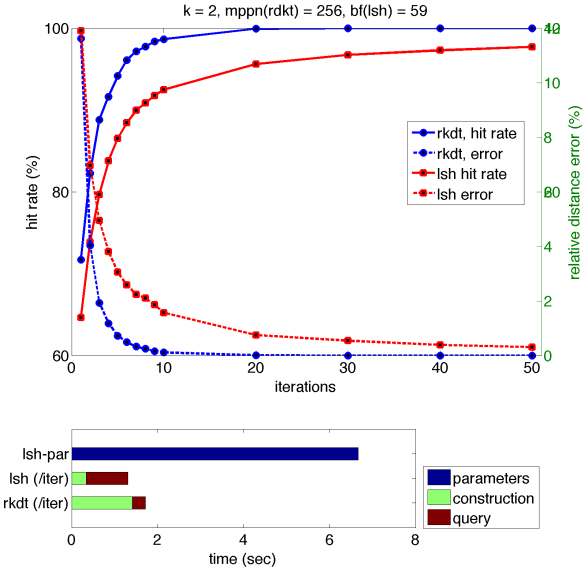
The accuracy and number of iterations for both LSH and RKD TREE depends on the intrinsic dimensionality of data. If the data has a low-dimensional parameterization, which is embedded into high dimensional feature space, our approach could converge in a small number of iterations. Comparing the two methods, the construction of LSH is usually faster than RKD TREE, but it requires more time to search for the nearest neighbors. The RKD TREE converges faster than LSH for points from MRI images and the dynamic system, but it converges slower for a normal distribution in high dimension. Figure **??** gives the hit rate and the time required by both approaches on the same datasets.

Another difference between RKD TREE and LSH is that the construction and query time of LSH depends both on the distribution of input data and the number of requested neighbors k. This is because LSH's performance depends on the search range for the query, which is relative to the degree of dispersion of data and k. The RKD TREE construction is independent of k and remains roughly the same for a fixed dimensionality. Due to LSH and RKD TREEhave different philosophy to group data points, they have different communication expense, which results in whether they have heavier construction portion or more expensive query part. LSH group points into buckets, and a query usually need to look at other buckets with a close hash value. However, the RKD TREE only visit on leaf node at each iteration. As a result, LSH usually search within more reference points. But unlike the RKD TREE, LSH only repartition points once (which is the most expensive communication part) at every iteration. Hence, it scales better than RKD TREE, which would repartition points at every level when we plant a tree. The main advantage of RKD TREE is that it has been demonstrated by several literatures, that the random projection tree do have the ability to find the intrinsic structure (low dimensional manifold embedded into the high dimensional data space), and as long as the intrinsic dimensionality of data is low, it can converge very fast[39, 72, 107].



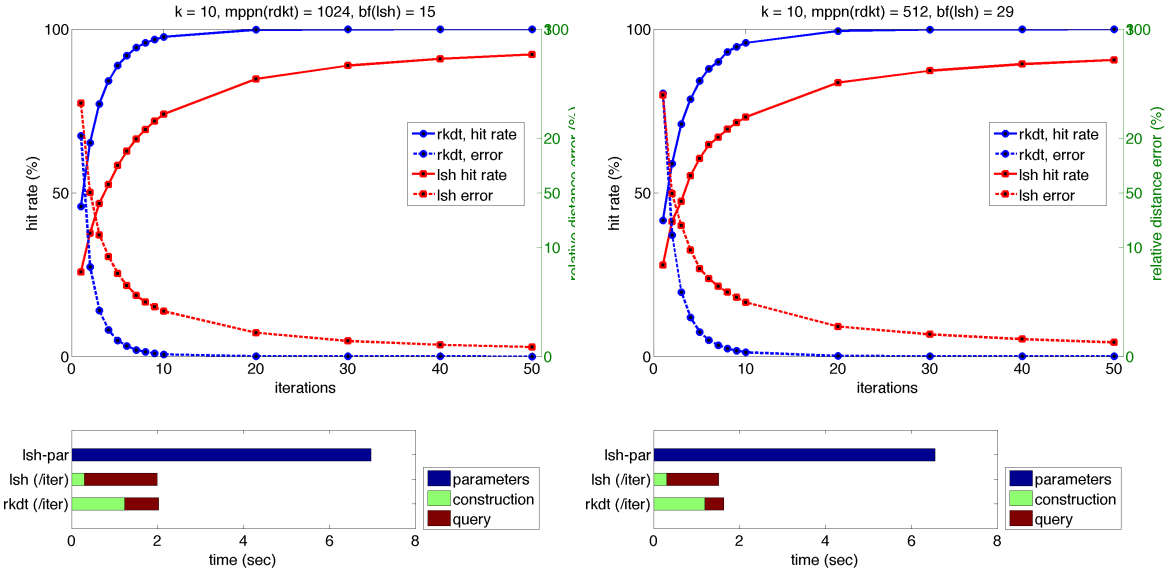
(a) mppn(rkdt) = 1024, bf(lsh) = 15

(b) mppn(rkdt) = 512, bf(lsh) = 29



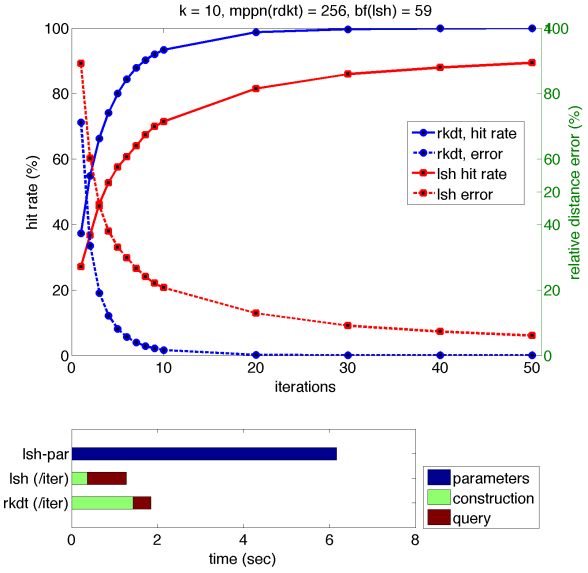
(c) mppn(rkdt) = 256, bf(lsh) = 59

Figure 18: Comparison between LSH and RKDT with $k = 2$: 480,000 Gabor feature points in total, 678 dimensional, 32 mpi process, each process per socket. 'bf' stands for the bucket numbers (bf*32) in lsh, 'mppn' is the maximum number of points per leaf node in rkdt. These parameters are chosen to make the bucket size and leaf node size are similar. Blue lines with circle marker are accuracies for 'RKDT', and red lines with square marker are accuracies for 'LSH'.



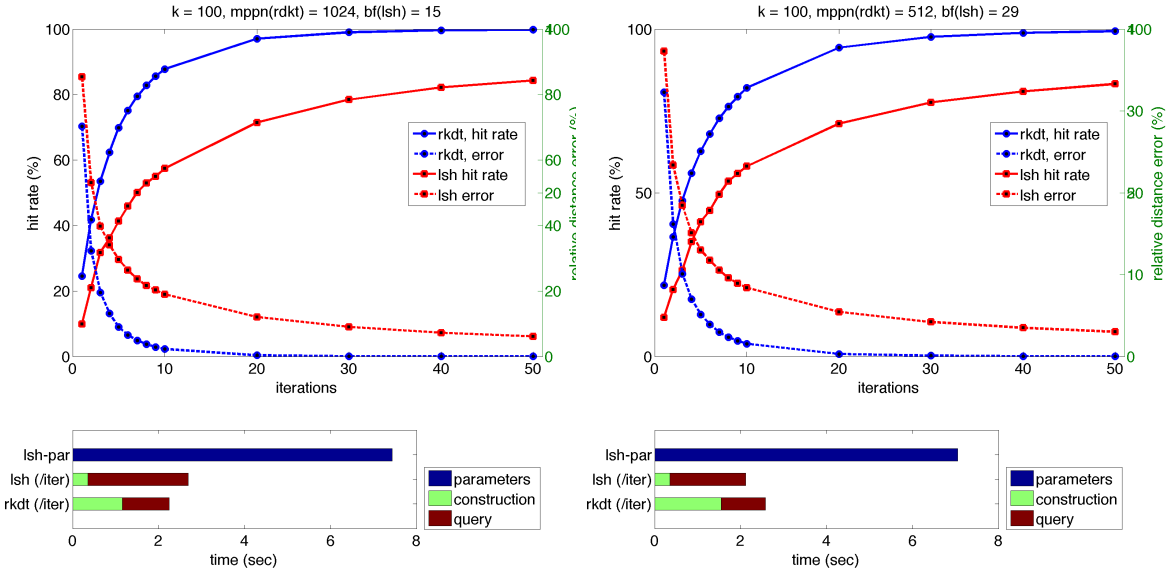
(a) mppn(rkdt) = 1024, bf(lsh) = 15

(b) mppn(rkdt) = 512, bf(lsh) = 29



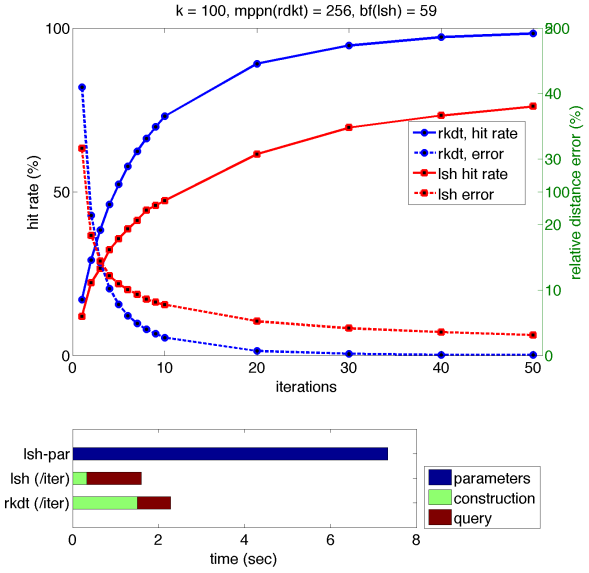
(c) mppn(rkdt) = 256, bf(lsh) = 59

Figure 19: Comparison between LSH and RKDT with $k = 10$: 480,000 Gabor feature points in total, 678 dimensional, 32 mpi process, each process per socket. 'bf' stands for the bucket numbers (bf*32) in lsh, 'mppn' is the maximum number of points per leaf node in rkdt. These parameters are chosen to make the bucket size and leaf node size are similar. Blue lines with circle marker are accuracies for 'RKDT', and red lines with square marker are accuracies for 'LSH'.



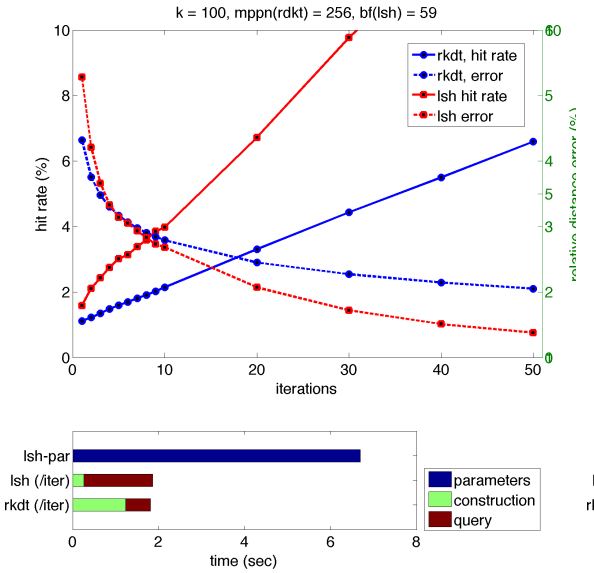
(a) mppn(rkdt) = 1024, bf(lsh) = 15

(b) mppn(rkdt) = 512, bf(lsh) = 29

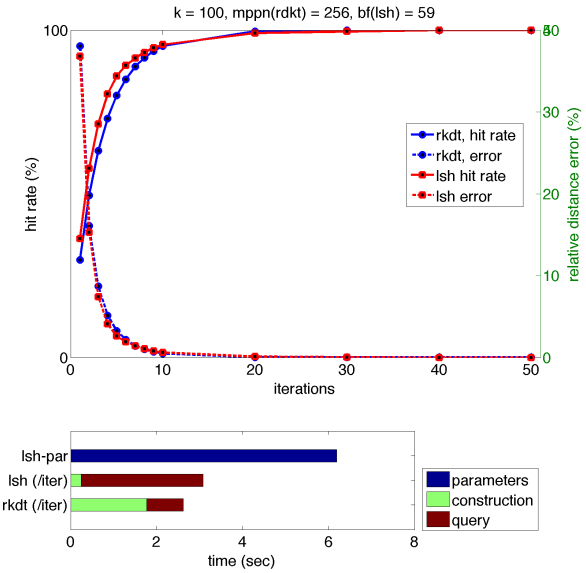


(c) mppn(rkdt) = 256, bf(lsh) = 59

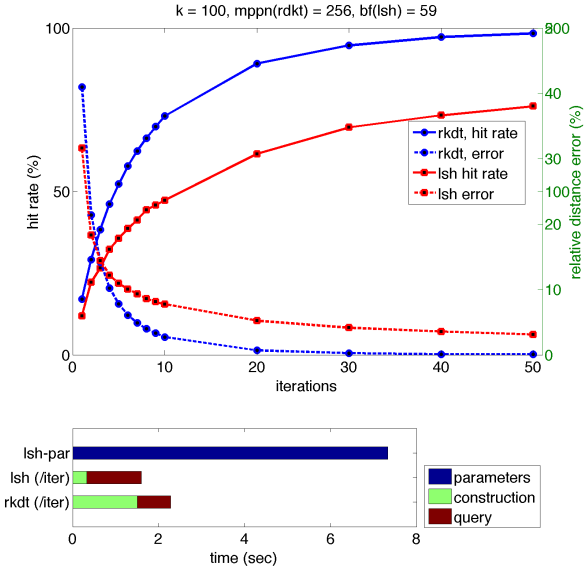
Figure 20: Comparison between LSH and RKDT with $k = 100$: 480,000 Gabor feature points in total, 678 dimensional, 32 mpi process, each process per socket. 'bf' stands for the bucket numbers (bf*32) in lsh, 'mppn' is the maximum number of points per leaf node in rkdt. These parameters are chosen to make the bucket size and leaf node size are similar. Blue lines with circle marker are accuracies for 'RKDT', and red lines with square marker are accuracies for 'LSH'.



(a) normal distribution



(b) embedded Gaussian (5d)



(c) Gabor features

Figure 21: Comparison between LSH and RKDT with $k = 100$ on three different data sets: normal distribution, embedded Gaussian and Gabor feature points. 480,000 points in total, 678 dimensional, 32 mpi process, each process per socket. 'bf' stands for the bucket numbers (bf*32) in lsh, 'mppn' is the maximum number of points per leaf node in rkdt. These parameters are chosen to make the bucket size and leaf node size are similar. Blue lines with circle marker are accuracies for 'RKDT', and red lines with square marker are accuracies for 'LSH'.

CHAPTER III

APPROXIMATE SKELETONIZATION KERNEL INDEPENDENT TREECODE

In computational statistics, kernelized methods require the rapid evaluation of kernel sums. One possible way to calculate the summation in Eqn. (2) is only considering the interaction between nearest points. In other words, the summation can be truncated as

$$\begin{aligned} \hat{f}_H(\mathbf{x}) = u(\mathbf{x}) &= \sum_{j=1}^{N} K(\mathbf{x}, \mathbf{x}_j) w_j \\ &\approx \sum_{j \in \mathcal{N}(x)} K(\mathbf{x}, \mathbf{x}_j) w_j \end{aligned} \tag{17}$$

where $\mathcal{N}(\mathbf{x})$ is the neighborhood of point $\mathbf{x}$. The drawback of direct truncation is compared to the radius of the neighborhood $\mathcal{N}(\mathbf{x})$, the bandwidth of kernels are relatively large so that the effects of points outside the neighborhood cannot be ignored. Especially for non-Gaussian kernels where the function values of kernels do not decrease drastically as the points are far away, the summation can introduce huge approximation errors.

In the case the far points cannot be ignored, we tried to use some compact representation to approximate the influence by compression. In this chapter, we present a fast algorithm for such sums and introduce novel methods for pruning and for approximating the far field. It is kernel-independent (does not use analytic expansions) and the pruning is combinatorial (not geometric). Its complexity depends linearly on the dimension. We present the structure of the algorithm in §3.2 and experimental results in §3.3 that demonstrate is performance. As a highlight, we report results for Gaussian kernel sums for one million points in 1000 dimensions and for problems in which the kernel has variable bandwidth.

3.1 Introduction

Considering the general *high dimensional N-body problem* of the form

$$u_i = u(\mathbf{x}_i) = \sum_{j=1}^{N} K(\mathbf{x}_i, \mathbf{x}_j) w_j, \quad \forall i = 1 \dots N. \tag{18}$$

We adopt a standard linear algebraic viewpoint and consider the problem of computing $\mathbf{u} = \mathbf{K}\mathbf{w}$ where $\mathbf{u}$ an $\mathbf{w}$ are N-dimensional vectors and $\mathbf{K}$ is a $N \times N$ matrix consisting of the pairwise kernel evaluations. The main idea in accelerating (18), which underlies powerful fast summation methods [22, 60], is to exploit low-rank blocks of the matrix $\mathbf{K}$ by constructing compact, approximate representations of them. These blocks are related to the smoothness of the underlying kernel function $K()$, which in turn is directly related to pairwise similarities between elements of a set, such as distances between points. Hierarchical data structures reveal these low rank blocks by rewriting (18) as

$$u_i = \sum_{j \in \mathrm{Near}(i)} K_{ij} w_j + \sum_{j \in \mathrm{Far}(i)} K_{ij} w_j, \tag{19}$$

where $\mathrm{Near}(i)$ is the set of points $\mathbf{x}_j$ whose contributions cannot be approximated well by a low-rank scheme and $\mathrm{Far}(i)$ indicates the set of points $\mathbf{x}_j$ whose contributions can.

Roughly speaking, fast summation algorithms for (18) can be categorized based on 1) how the Near(i) and Far(i) splittings are defined, and on 2) the compact representation used to approximate K_{ij} (for $j \in \mathrm{Far}(i)$). Given these two ingredients, the data are partitioned with a space partitioning tree. Then, to evaluate the sum for a query, we can use a *treecode* algorithm. We split nodes of the tree into Near and Far sets. The contributions from Near nodes are evaluated directly, and those from Far nodes are approximated from their compact representations.

3.1.1 Our approach and contributions

We present "ASKIT", a fast kernel summation treecode with a **new near-far** field decomposition and a new compact representation for the **far field**.

• *Far field*: Our compact representation of the far field contribution uses an approximate interpolative decomposition (ID) (for the exact ID see [83, 64]) which uses nearest-neighbor information for sampling. Our method enjoys several advantages over existing methods: it only requires kernel evaluations, rather than any prior knowledge of the kernel such as in analytic expansion-based schemes; it can evaluate kernels which depend on local structure, such as kernels with *variable bandwidths*; and its effectiveness depends only on the linear algebraic rank of sub-blocks of the kernel matrix and provides near-optimal (compared to SVD) compression without explicit dependence on the ambient dimension of the data.

• *The near field-far field decomposition (pruning)* is *not* done using the usual distance/bounding box calculations. Instead, we use a combinatorial criterion based on nearest neighbors. Experimentally, this approach improves pruning in high dimensions and opens the way to more generic similarity functions. Also, based on this decomposition we derive sharp complexity bounds for the overall algorithm. This method also lends itself to more scalable parallel implementations. Our scheme depends on the algebraic properties of the kernel matrix which encapsulate both the kernel and the structure of the dataset, for example low intrinsic dimensionality that is not necessary flat or manifold.

Our goal in this chapter is to outline the new algorithm and provide experimental evidence for the feasibility of this approach. There remain multiple open questions on error, complexity, parameter selection, robustness, and extensibility that we do not address here. We present results on synthetic datasets for Gaussian kernels, with both fixed and variable (source-dependent) bandwidths. Finally, we present results on distributed memory architectures. We thus demonstrate the feasibility of our novel approach. We demonstrate the linear-dependence on the ambient dimension by conducting an experiment in 1000 dimensions in which the far field can not be truncated and for which we achive six digits of relative accuracy and a 20$\times$ speedup over a highly optimized direct, N^2, evaluation. On a 5M-point, 18D UCI dataset we obtain 25$\times$ speedup. On a 5M-point, 128D syntehtic

dataset we obtain 2000$\times$ speedup using 256 dual-scocket nodes.

One commonly used kernel in statistical learning is the Gaussian kernel $K(\mathbf{x}_i - \mathbf{x}_j) = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/\sigma_j^2)$, and we focus our experiments on this kernel. We allow for a bandwidth that depends on the source point j. This is the variable bandwidth case.

We propose a new distributed memory tree construction that does not replicate the tree and allows highly-nonuniform distribution of points by ensuring load balancing. Here we outline the basic features of the new algorithm and we present experimental evidence on the scalability of the method in high-dimensional datasets. Our implementation combines the Message Passing Interface (MPI) protocol for high-performance distributed memory parallelism and the OpenMP protocol for shared memory parallelism.

In our experiments we use Euclidean distances and classical binary space partitioning trees. The nearest neighbors are computed by random projection trees described in chapter 2.

3.1.2 Related work

Throughout, we refer to a point $\mathbf{x}_i$ for which we compute a u_i as a *target* and a point $\mathbf{x}_j$ as a *source*. We fix a group of source points and use K to represent the interaction of these source points with distant targets (here we abuse the notation, since K is just a block of the original matrix). The compact representation used in treecodes consists of a factorization $K \approx LM$, where both L and M are low rank. We divide previous approaches broadly into two categories. Analytic expansions construct the factors L and M formally from the kernel function $K()$. Classical work in this category in two and three dimensions includes [22] and [60]. In higher dimensions related work includes [61, 59, 76, 92, 134]. One of the fastest schemes in high dimensions is the improved fast Gauss transform [134, 92].The the analytic expansions scale either as $\mathcal{O}(c^d)$ (resulting in SVD-quality errors) or as $\mathcal{O}(d^c)$, where $c > 1$ is related to approximation order. In addition to the expensive scaling with ambient dimension, the approximations in these methods must be derived and implemented

individually for each new kernel function.

Kernel-independent methods construct L and M using evaluations of $K()$. Examples include [135, 58] that scale as $\mathcal{O}(c^d)$. Other methods compute approximation representations on the fly using Monte Carlo methods [77]. While this scheme depends only on the intrinsic dimensionality of the data, Depending on $K()$, the factors L and M can be constructed analytically and perform as well as the SVD which the best that can one hope for standard error measures. , and we classify them as kernel-independent in the sense that making them work for a new kernel is immediate, as long as a low-rank factorization is possible. In essence, kernel independent methods sample K to construct L and M. A treecode in which the far-field is sampled using Monte-Carlo was presented in [77]. This is the only scheme that we know of that depends only on the intrinsic dimension. However, it does not construct a representation of the far-field contribution, it does not exploit the low rank structure of K explicitly and this causes it to converge slowly. Another scheme that requires kernel evaluations only (in the frequency domain) is [108]; its performance depends on the ambient dimension and on the kernel being diagonalizable in Fourier space. Work on kernel summations in distributed memory includes [78]. Finally, since we present results on distributed memory, let us mention the work of [78]. Our scheme is quite different but the specifics will be reported elsewhere since the parallelization of the scheme is not the main point of this contribution.

3.2 Algorithms

We now turn to the description of our new algorithm. We begin by surveying the basic outline of a treecode. First, we construct a space-partitioning tree on the data. We use this term to broadly cover any hierarchical partitioning of the data set such that nearby (or similar) points are grouped together. For our purposes, a tree consists of internal nodes with two or more children and leaf nodes with no children. The binary tree we used is the same as in the RKD TREE in chapter 2. We split a node by estimating the farthest pair of

Table 16: Here we summarize main **notation** used in the text. In addition to the information above, we use $\cup$ to indicate the set union of two index sets, $\setminus$ the difference of two index sets, $|\cdot|$ the number of elements in set. We define ISLEAF(α) to be `true` if α is a leaf. We also use $w(\mathcal{I})$ to indicate the components of vector w determined by an index set $\mathcal{I}$ and we use a similar notation for matrices.

N	number of points
K	the kernel function or matrix
κ	number of nearest neighbors per point
m	number of points per leaf
s	number of skeleton points
$\mathcal{X}$	$\{1, \ldots, N\}$
$\mathcal{X}_\alpha$	ids of points in α

α	tree node or simply node
α_i	leaf node that contains point i
$\mathcal{N}_i$	neighbor list for point i
$\mathcal{N}_\alpha$	neighbor list for node α
$\mathcal{A}_\alpha$	ancestor list of node α
$\mathtt{r}(\alpha)$	right child of node α
$\mathtt{l}(\alpha)$	left child of node α

points in it. Then we project all points onto the line between these points, and split at the median. The splitting planes are shown in Figure 7.

Given such a tree, treecodes perform a two-step process to approximate the kernel summations. First, in the a bottom-up tree traversal (also known as the upward pass), we create a compact representation at each node due to all the points in it. We form these representations at the leaves, then pass them up to parents and combine them. Second, in a top-down traversal (also known as the downward pass), we use these representations to compute approximate potentials. For each point, we traverse the tree from the top down. At a node, we apply a pruning criterion to determine whether to approximate or not. If we can approximate, then we apply the compact representation to account for the influence of all the points in the node and *prune*. If not, then we proceed to the children. If we still cannot prune at a leaf, we evaluate the contribution of the leaf's points directly. In the remainder of this section, we describe in detail how we carry out each of these steps in ASKIT. We begin with the construction of compact representations for tree nodes.

3.2.1 Interpolative Decompositions and Sampling.

The key component of our method is the representation of the far field generated by source in a node using an approximate ID scheme. Let $K \in \mathbb{R}^{n \times m}$. Let $\mathcal{S}$ be an index set with $|\mathcal{S}| = s$ and $1 \leq \mathcal{S}_j \leq m$. Let $K_S = K(:, \mathcal{S})$ be the columns of K indexed by $\mathcal{S}$ and K_R be the remaining *unskeletonized* columns of K. Assuming $s < m < n$ and that K_S is full

rank, we can approximate the columns of K_R by $K_S P$ where $P = K_S^\dagger K_R, P \in \mathbb{R}^{s \times m}$. The ID consists of the index set $\mathcal{S}$, referred to as the *skeleton*, and matrix P. Following [83], we refer to the construction of this approximation for a matrix as *skeletonization*.

In order to compute an ID such that $\|K_R - K_S P\|$ is small, we employ a pivoted QR factorization to obtain $K\Pi = QR$ for some permutation Π, an orthonormal matrix Q, and upper triangular matrix R. The skeleton $\mathcal{S}$ corresponds to the first s columns of $K\Pi$, and the matrix P can be computed from R in $\mathcal{O}(s^3 + s^2(m-s))$ time. It can be shown [64] that

$$\|K_R - K_S P\| \leq \sqrt{1 + m(m-s)}\sigma_{s+1}(K), \tag{20}$$

where σ_{s+1} is the i^{th} singular value of K. The overall cost for $s < m < n$ is $\mathcal{O}(nm^2)$.

Far field using skeletonization: We will use ID to compactly represent the far field of a leaf node α. Let $\mathcal{X}_\alpha$ be the set of points assigned to α (assume $|\mathcal{X}_\alpha| = m$). Let $\mathbf{K}_\alpha := K(\mathcal{X} \setminus \mathcal{X}_\alpha, \mathcal{X}_\alpha) \in \mathbb{R}^{(N-m)\times m}$. Also let $w_\alpha = w(\mathcal{X}_\alpha) \in \mathbb{R}^m$. Our task is to construct an approximation to $\mathbf{K}_\alpha w_\alpha$. We choose s, compute the skeleton $\mathcal{S}_\alpha$ of $\mathbf{K}_\alpha$ and set

$$\mathbf{K}_\alpha w_\alpha \approx \tilde{\mathbf{K}}_\alpha \tilde{w}_\alpha, \text{ where } \tilde{\mathbf{K}}_\alpha := \mathbf{K}_\alpha(:, \mathcal{S}_\alpha),\ \tilde{w}_\alpha := w_\alpha(\mathcal{S}_\alpha) + P_\alpha w_\alpha(\mathcal{R}), \tag{21}$$

and $\mathcal{R}$ is the index set of the unskeletonized columns of $\mathbf{K}_\alpha$.

Given the skeleton $\mathcal{S}_\alpha$ and skeleton weights $\tilde{w}_\alpha$, we can efficiently approximate the contribution to some target point $\mathbf{u}_i$. We first compute the $1 \times s$ matrix of interactions $K(\mathbf{u}, \mathcal{S}_\alpha)$, then apply it to the vector $\tilde{w}_\alpha$, to get an approximation with error bounded by (20).

This approach leaves two issues unanswered. First, how do we choose s? We discuss this in §3.2.3. Second, computing the ID is more expensive than directly evaluating $\mathbf{K}_\alpha w_\alpha$. We address this point next.

Approximate skeletonization. We have $\mathcal{O}(N/m)$ leaves and the skeletonization of each leaf costs $\mathcal{O}(Nm^2)$. When performed for all leaf nodes, this will require $\mathcal{O}(N^2 m)$ work. Instead, we will compute the skeleton of a smaller matrix, which has only a random

subset of the rows of $\mathbf{K}_\alpha$. That is, we select ℓ rows with $m < \ell \ll N$ and whose index set we denote by $\mathcal{T}_\alpha$. We form $\mathbf{K}_\alpha(\mathcal{T}_\alpha, :) \in \mathbb{R}^{\ell \times m}$, and compute the skeleton $\mathcal{S}_\alpha$ of size s and the corresponding projection matrix P_α. This is equivalent to choosing s target points. The complexity of the construction for one leaf becomes $\mathcal{O}(\ell m^2)$; thus the overall complexity for all nodes in the tree becomes $\mathcal{O}(N\ell m)$.

Sampling rows of K: We need to choose a small number of rows such that the ID of $\mathbf{K}_\alpha(\mathcal{T}_\alpha, :)$ will be close to the ID of K. Randomized linear algebra algorithms can achieve this by either random projections [64] or the construction of an importance sampling distribution [87]. Either approach requires $\mathcal{O}(N)$ work per node. However, for smoother kernels that decay with distance (or, more generally, dissimilarity) the nearest (more similar) points will tend to dominate the sum in (19). Following this intuition, if we can include the nearest neighbors of each point in $\mathcal{X}_\alpha$, then we expect this to be a reasonable approximation to an importance sampling distribution. If we do not have enough neighbors, we add additional uniformly chosen points to reach a sufficient sample size. This process is discussed below.

3.2.2 The Treecode

Using the ID as a compact representation, we can now describe the main steps of our treecode: (i) Approximate the κ-**nearest neighbors** for all $\mathbf{x}_i$. (ii) Top-down **binary tree decomposition** for $\mathcal{X}$. (iii) Bottom-up traversal to build **neighbor lists** for interior (non-leaf) nodes. (iv) Bottom-up traversal to compute **skeletons and equivalent weights**. (v) Top-down traversal to **evaluate** u_i at each point i. We describe the individual steps below in detail. *(See also the supplementary material for figures and more detailed explanation.)*

Nearest neighbors and binary tree decomposition: To find nearest neighbors we use a greedy search using random projection trees [38]. We build a tree and for each $\mathbf{x}_i$ we collect κ-nearest neighbors found by exhaustive search among the other points in the leaf node that contains $\mathbf{x}_i$. Then we discard the tree (we do not perform top-down searches) and iterate, keeping the best candidate neighbors found at each step. The binary tree used in

the treecode is built using the following rule: to split a node α, we compute its center ($\mathbf{x}_c$), then the furthest point to $\mathbf{x}_c$ ($\mathbf{x}_l$), then the furthest point to $\mathbf{x}_l$ ($\mathbf{x}_r$). We project all the points on the line $(\mathbf{x}_l, \mathbf{x}_r)$, compute the median, and split them into two groups. We recurse until every leaf gets no more than m points.

Node neighbor lists: During the skeletonization, we need to sample the far field. To do this we need to construct *node neighbor* lists. These lists are defined in Algorithm 14 and are constructed in a bottom-up fashion using a standard preorder traversal of the tree. The

Algorithm 14 BUILDNEIGHBORS(α)

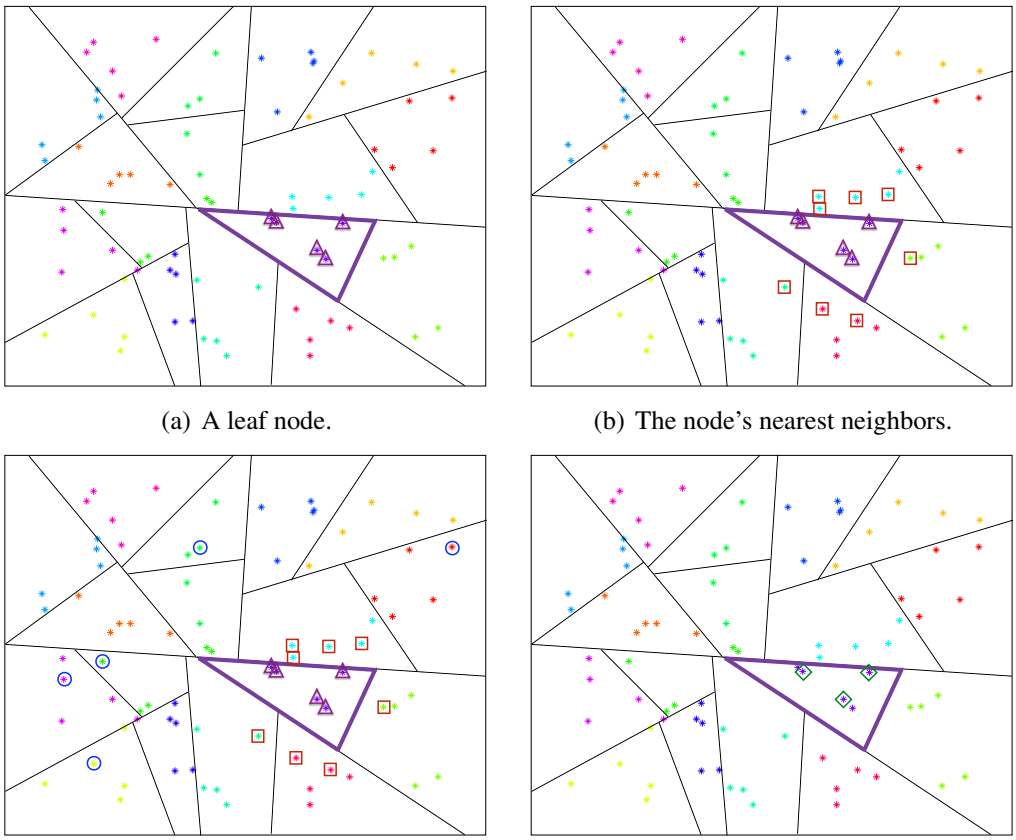
1: `if` ISLEAF(α), $\mathcal{N}_\alpha := (\cup_{i \in \mathcal{X}_\alpha} \mathcal{N}_i) \setminus \mathcal{X}_\alpha$
2: `else` $\mathcal{N}_\alpha := \left(\mathcal{N}_{\mathtt{r}(\alpha)} \cup \mathcal{N}_{\mathtt{l}(\alpha)}\right) \setminus \left(\mathcal{X}_{\mathtt{r}(\alpha)} \cup \mathcal{X}_{\mathtt{l}(\alpha)}\right)$

set-difference operations can be done in $\mathcal{O}(1)$ time per point using the binary-tree Morton id of every node and every point. The Morton id is a bit array that codes the path from the root to the node or point. The Morton id of a point is the Morton id of the leaf node it belongs to.

Skeletonization of leaves: Let $\mathcal{X}_\alpha$ be the set of points. Let the contribution of this node to all $\mathcal{X} \setminus \mathcal{X}_\alpha$ be denoted as $\mathbf{K}_\alpha$. The goal is to approximate $\mathbf{K}_\alpha$. Standard methods using analytic expansions the create a separable approximation, which in algebraic terms can be write as $\mathbf{K}_\alpha = \mathbf{L}\mathbf{M}_\alpha$, where both $\mathbf{L}$ and $\mathbf{M}_\alpha$ are low rank. Instead we use an interpolative decomposition. But to compute the interpolative decomposition requires computing $\mathbf{K}_\alpha$ and doing so for every leaf node results in $\mathcal{O}(N^2)$ work since $\mathbf{K}_\alpha$ is a $N \times |\mathcal{X}_\alpha|$ matrix. Instead we need to sample the rows of $\mathbf{K}_\alpha$. To do so we select a set of targets $\mathcal{T}_\alpha$, we construct $\mathbf{K}(\mathcal{T}_\alpha, \mathcal{X}_\alpha)$, compute each skeletonization. As noted above, we will approximate the ID of $\mathbf{K}_\alpha$ by computing the ID of a matrix $\mathbf{K}(\mathcal{T}_\alpha, \mathcal{X}_\alpha)$ for some small set of rows $\mathcal{T}_\alpha$. Following our intuition that nearby points dominate the interaction, we choose the nearest neighbors of the points in $\mathcal{X}_\alpha$ which are not themselves in $\mathcal{X}_\alpha$ as a starting point and include uniformly sampled distant points as needed:

$$\mathcal{T}_\alpha = \mathcal{N}_\alpha \cup \text{SAMPLE}(\, \mathcal{X} \setminus (\mathcal{X}_\alpha \cup \mathcal{N}_\alpha),\ \ell - |\mathcal{N}_\alpha|\,), \tag{22}$$

That is, we randomly sample $\ell - |\mathcal{N}_\alpha|$ points excluding the points in α and we use these points along with the neighbors $\mathcal{N}_\alpha$ as target points. We then compute the ID of $\mathbf{K}(\mathcal{T}_\alpha, \mathcal{X}_\alpha)$ to obtain the skeleton $\mathcal{S}_\alpha$ and skeleton weights $\tilde{w}_\alpha$ for $\mathbf{K}_\alpha$.



(a) A leaf node. (b) The node's nearest neighbors.

(c) Sampling distant points. (d) The resulting skeleton.

Figure 22: Skeletonizing a leaf. Fig. 22(a). We highlight the leaf node to be skeletonized. The points to be approximated are shown with triangles. **Fig. 22(b).** We compute the union of the lists of nearest neighbors of the points in the leaf, and exclude points that belong to the leaf itself. These points are highlighted with squares. **Fig. 22(c).** We sample additional distant points for the skeletonization. These points are highlighted with circles. We compute the matrix of interactions with rows given by the neighbors and samples (squares and circles) and columns given by the points in the leaf (triangles). **Fig. 22(d).** We compute the ID of this matrix to obtain the skeleton points, highlighted with diamonds. We can now approximate the contribution of the points in this node to a distant target point using only the interactions with these skeleton points.

Skeletonization of internal nodes. To build the far field approximation for an interior node α we use the same algorithm. Instead of using all the points in the leaf descendants of α, we use the combined skeleton points $\mathcal{S}_{\mathtt{r}(\alpha)} \cup \mathcal{S}_{\mathtt{l}(\alpha)}$ and the neighbors list $\mathcal{N}_\alpha$ constructed

Algorithm 15 SKELETONIZE(α)

1: `if` $\neg$ ISLEAF(α)
2: SKELETONIZE($\mathtt{r}(\alpha)$), SKELETONIZE($\mathtt{l}(\alpha)$)
3: $\mathcal{X}_\alpha = \mathcal{S}_{\mathtt{r}(\alpha)} \cup \mathcal{S}_{\mathtt{l}(\alpha)}$
4: Create sampling targets using (22)
5: Skeletonize $\mathcal{X}_\alpha$ using QR factorization and store $\mathcal{S}_\alpha$ and $\tilde{w}_\alpha$

with BUILDNEIGHBORS(α). Let $\tilde{w}_{\mathtt{l}(\alpha)}$ and $\tilde{w}_{\mathtt{r}(\alpha)}$ be the vectors of skeleton weights of the children.

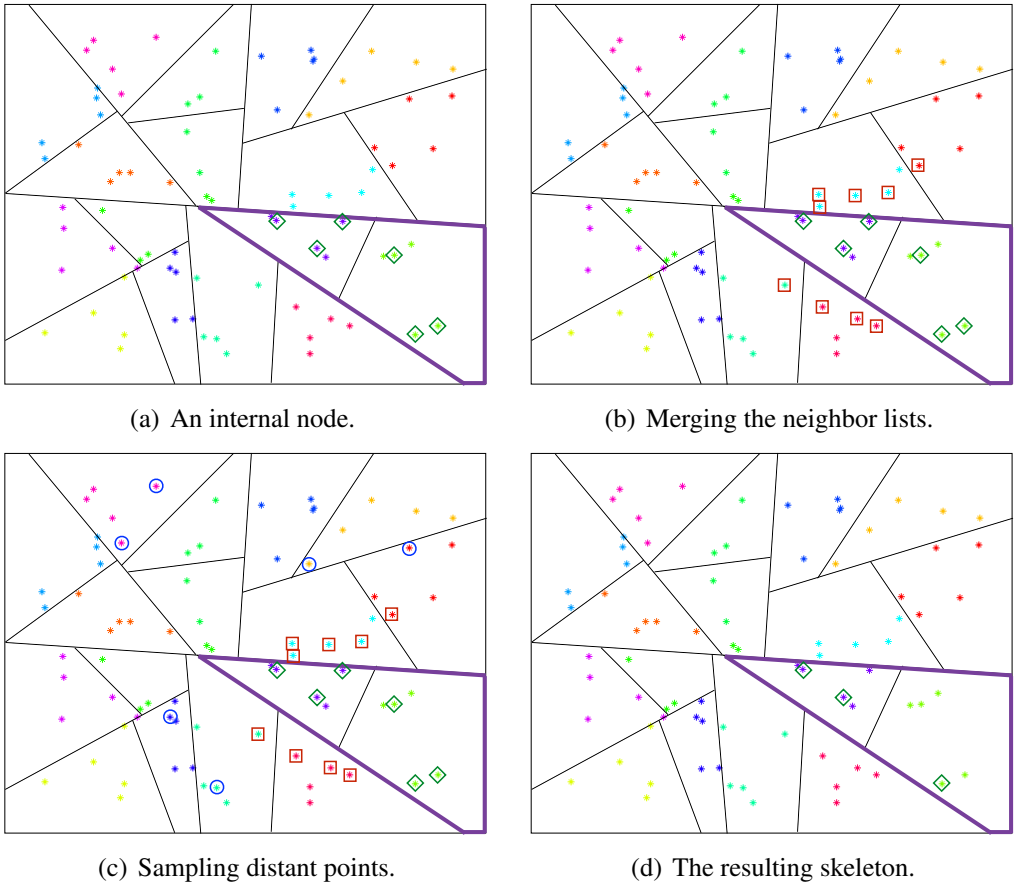
We compute the ID of the matrix $\mathbf{K}(\mathcal{T}_\alpha, \mathcal{S}_{\mathtt{r}(\alpha)} \cup \mathcal{S}_{\mathtt{l}(\alpha)})$ to obtain a skeleton for α and P_α. We then apply P_α to the unskeletonized part of the concatenation of $\tilde{w}_{\mathtt{l}(\alpha)}$ and $\tilde{w}_{\mathtt{r}(\alpha)}$ to obtain the skeleton weights. These ideas are summarized in SKELETONIZE(α).

We also require the ability to efficiently combine the representations of two child nodes. Let S be a node and L and R its children, with skeletons and effective charge vectors. Merge the skeletons of L and R into a single set of $2k$ points and concatenate the two effective charge vectors into a single $2k$ vector ϕ. Since the set of points for which the outgoing representation of S must hold is a subset of the sets for which L and R hold, this will be a correct ID for S. However, we can still reduce the size of this representation. We compute an ID of the matrix K whose columns correspond to the $2k$ combined skeleton points and whose rows are the distant points for which the representation needs to hold. We then compute a rank k ID of this matrix to obtain a subset of k columns and a $k \times 2k$ projection matrix. We apply this matrix to the effective charges. The overall skeletonization is summarized in Algorithm 15.

Pruning: During evaluation phase we use a standard top down traversal. For every $\mathbf{x}_i$, we start at the root and traverse the tree. The pruning is based on the *neighbors* of $\mathbf{x}_i$ and has nothing to do with distance or kernel evaluations. Node α is **not pruned** if it is either an ancestor of $\mathbf{x}_i$ or it is an ancestor of *any* of the nearest neighbors of $\mathbf{x}_i$:

$$\text{PRUNE}(\alpha, i) = \text{ISTRUE}(\nexists j \in \{i \cup \mathcal{N}_i\} : \alpha \in \mathcal{A}_j) \tag{23}$$

Note that the ancestor check can be done in $\mathcal{O}(1)$ time using Morton ids.



(a) An internal node. (b) Merging the neighbor lists.

(c) Sampling distant points. (d) The resulting skeleton.

Figure 23: Skeletonizing an internal node. Fig. 23(a). We skeletonize the highlighted internal node. The skeletons of its children are highlighted with diamonds. **Fig. 23(b).** We merge the nearest neighbor lists of the children, then exclude all of the points belonging to the node to be skeletonized. These points are highlighted with squares. **Fig. 23(c).** We sample additional distant points, highlighted with circles. We compute the matrix of interactions with rows given by the neighbors and samples (squares and circles) and columns given by skeletons of the child nodes (triangles). **Fig. 23(d).** We compute the ID of this matrix to obtain a skeleton for the parent node, which is a subset of the combined skeletons of the children.

Algorithm 16 $u_i = \text{EVALUATE}(\mathbf{x}_i, \alpha)$

1: `if` PRUNE(α, i), `return` $K(\mathbf{x}_i, \mathcal{S}_\alpha)\tilde{\mathbf{w}}_\alpha$ {Approximate ((23))}
2: `if` ISLEAF(α), `return` $K(\mathbf{x}_i, \mathcal{X}_\alpha)\mathbf{w}_\alpha$ {Direct evaluation}
3: `return` EVALUATE$(\mathbf{x}_i, \mathtt{r}(\alpha))$ + EVALUATE$(\mathbf{x}_i, \mathtt{l}(\alpha))$ {Recursion}

Algorithm 17 $\mathbf{u}$=ASKIT$(\mathcal{X}, \mathbf{w}, s, \ell, m, \mathcal{N}(\mathcal{X}))$

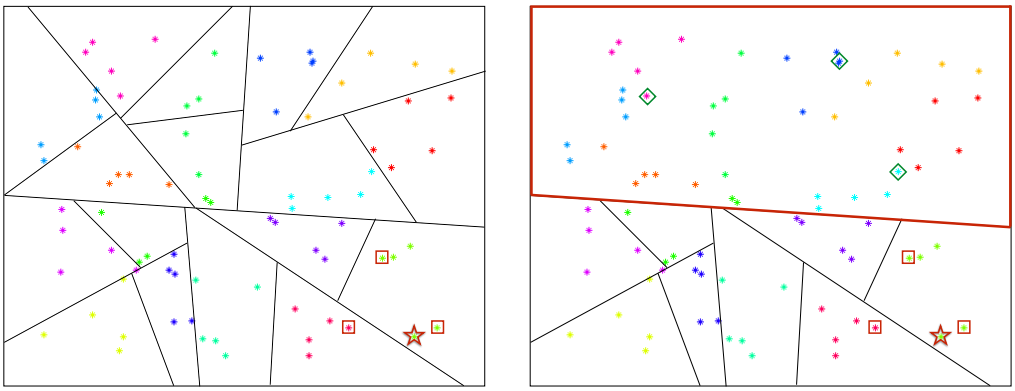
1: α =BINARYTREE$(\mathcal{X}, m)$ {Build binary tree, α is the root}
2: BUILDNEIGHBORS(α) {Bottom-up traversal}
3: SKELETONIZENODE(α) {Bottom-up traversal}
4: $u_i = \text{EVALUATE}(\mathbf{x}_i, \alpha) \quad \forall i \in \mathcal{X}$ {Top-down traversal}

The evaluation algorithm and the overall scheme are summarized in Algorithm 16 and Algorithm 17 respectively. Figure 24 illustrates the steps of evaluating the approximate summations. **Fig. 24(a).** We evaluate the potential at a target point, highlighted with a star. We show its nearest nearest neighbors as well, highlighted with squares. **Fig. 24(b).** We traverse the tree in a top down fashion. We can prune a node if the node does not contain any neighbors of the target point. The node highlighted in red satisfies our pruning criterion, so we evaluate its contribution approximately. We compute the interactions between the skeleton points (diamonds) and the target. We use the skeleton weights for these points to compute the effective contribution of the node. **Fig. 24(c).** We continue traversing the tree. Once again, we have a node that contains no nearest neighbors of the target, so we use the skeleton points to compute its approximate contribution. **Fig. 24(d).** We still use the approximate representation at leaves which satisfy the pruning criterion. **Fig. 24(e).** We continue the approximate evaluation of all nodes that satisfy the pruning criterion. **Fig. 24(f).** The leaves highlighted in blue do not satisfy the pruning criterion, so we compute their contribution directly. We evaluate the direct interaction between all points in these nodes (highlighted with squares) and the target point.

3.2.3 Complexity and error

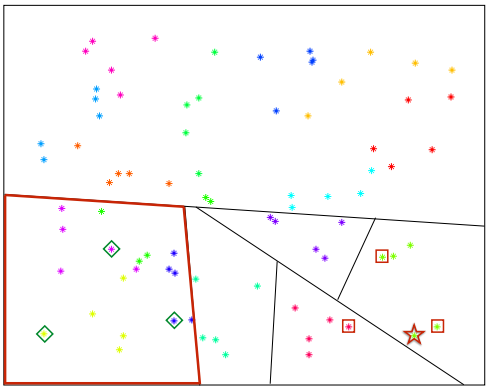
Our proposed method has the following **parameters**: • m: the number of points per leaf node; it controls the error and the runtime since it governs the trade-off between near and far interactions. • κ: the number of nearest neighbors for sampling and pruning. The larger κ the less we prune. The larger κ the better our sample is when we compute the ID. In our tests, we set $\kappa = 2m$. • s: the skeleton size. The higher s, the more accurate and expensive the skeletonization and evaluation phases are. One could use $\|K_R - K_S P\|$ and choose s adaptively. • ℓ: the row sampling size for $\mathbf{K}_\alpha$. Larger values allow a more accurate ID but slower skeletonization. We require $\ell > m$ so that ID problem is overdetermined.

Complexity: We now turn to the computational complexity of our algorithm. The number of leaves is $M = N/m$ and the total number of nodes is $2M - 1 = \mathcal{O}(M)$. It

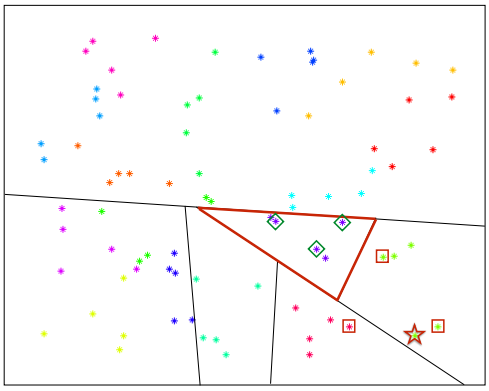


(a) Evaluating at a target point (star).

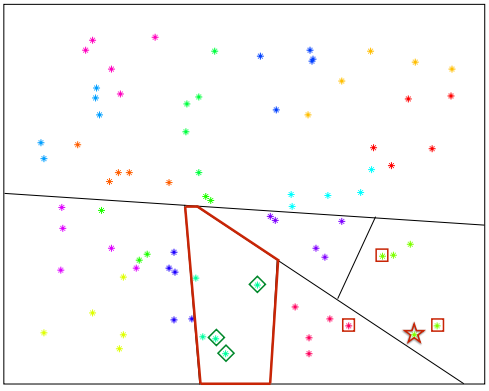
(b) Pruning the highlighted node.



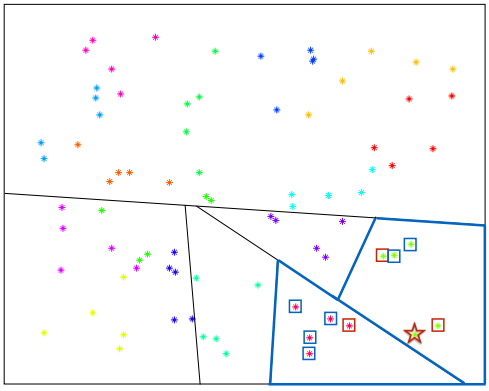
(c) Pruning another node.



(d) Pruning a leaf node.



(e) Pruning another leaf node.



(f) Directly evaluating un-prunable leaves.

Figure 24: Evaluating the approximate summation.

is easy to show that the cost of building the neighbor lists is $\mathcal{O}(M\kappa \log \kappa)n$. The cost of skeletonization is $\mathcal{O}(M\ell m^2)$. The cost of the downward pass depends on our ability to prune. Given a target point x_i, let ξ be the number of nodes we visit to approximate u_i. We decompose ξ into two parts: ξ_n for nodes evaluated directly and ξ_f for nodes approximated via the skeleton ($\xi = \xi_f + \xi_n$). Given these parameters, the cost of the downward pass for point x_i is at most $\xi_n m$ for direct evaluation plus at most $\xi_f s$ for approximations. Taking the worse case values of ξ_n and ξ_f for all evaluation points, the overall downward pass cost is bounded by $N(\xi_n m + \xi_f s)$.

We now bound ξ_n and ξ_f. For ξ_n the worst case is that for every point we visit κ different leaves, (since we use the points nearest neighbors for pruning). Therefore $\xi_n \leq \kappa$. To bound ξ_f we proceed as follows. Given point a target point i, we cannot prune nodes in $\mathcal{A}(\mathcal{N}_i)$. The worst case is that all nodes in $\mathcal{A}(\mathcal{N}_i)$ are unique. Since $|\mathcal{N}_i| = \kappa$, $|\mathcal{A}(\mathcal{N}_i)| \leq \kappa \log M$. But if these nodes are unique, that means we can prune their siblings by using their skeletonization. Thus, $\xi_f = \mathcal{O}(\kappa \log M)$. So the costs of skeletonization T_S and evaluation T_E are bounded *in terms of kernel evaluations* by

$$T_S = \mathcal{O}\left(N\ell m\right) \quad \text{and} \quad T_E = \mathcal{O}\left(Nm\,\kappa + Ns\,\kappa \log \frac{N}{m}\right). \tag{24}$$

The ambient dimension d enters only in the kernel evaluations, which can be highly optimized. For the Gaussian kernel this results in skeletonization cost that is dT_S and evaluation cost that is dT_E.

Error: There are two sources of error in approximating $\mathbf{K}_\alpha$: the use of the ID in place of the full kernel matrix (forming the skeleton), and the selection of a subsample of the rows for the computation of the ID (the sampling). Sampling rows and columns of a matrix is a widely studied problem. It can be shown that a carefully constructed importance sampling distribution based on statistical leverage scores can provide a $(1+\zeta)\sigma_{s+1}$ reconstruction error if the sample size ℓ is proportional to $s \log s$ [87]. Creating such an importance distribution is at least as expensive as the computation of the kernel summation exactly. However, we employ a heuristic choice of nearest neighbors and uniformly chosen

samples. We restrict our discussion to the case where we achieve this error and leave a more detailed analysis of this part of the error to future work.

The remaining error from the ID is given by (20). In the worst case, for a given evaluation point, we incur this error each time we prune. We know that the number of prunes is bounded by $\kappa \log \frac{N}{m}$. Denoting by $\mathcal{V}_i$ the nodes whose skeletonization was used to evaluate the potential at $\mathbf{x}_i$, the overall absolute error ϵ is then bounded by

$$\epsilon = \max_i |u_i - u_{\text{exact}}(\mathbf{x}_i)| \leq \kappa \log \frac{N}{m} \left((1 + \sqrt{m} + \zeta) \max_{\alpha \in \mathcal{V}_i} \sigma_{s+1}^{\alpha} \right) \|w\|. \tag{25}$$

We remark that the ambient dimension does not appear in the error estimate; κ and errors in finding the exact nearest neighbors affect the maximum of σ_{s+1}^{α}; ζ depends on ℓ and the sampling scheme.

3.3 Experimental results

Setup: In Table 18, we report preliminary results that show the feasibility of ASKIT. All code has been implemented in C++. The direct evaluation is highly optimized. We use the Intel MKL for linear algebra, and use OpenMP and the MPI parallelism. The tests were performed on a infiniband cluster with dual-socket 2.8GHz Intel Xeon E5-2680 nodes. To test the code, we use normally distributed points in 4, 16, and 64 dimensions for which the intrinsic and ambient dimension coincide. We present results for 100K and 1M points. We also consider the embedding in 1000D of a set of points normally distributed in a 4D hypersphere. We also used a UCI dataset (SUSY [7]) with 5M points in 18 dimensions. The sources and targets coincide and we report timings for all pairwise interactions. We present wall-clock times for finding the neighbors, constructing the skeletonization, and the evaluation.

The performance of our nearest neighbor search affects the overall runtimes and the performance of ASKIT. For this reason, we report the hit rate accuracy (percentage of correct neighbors). We present results for the Gaussian kernel $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/h_j^2)$ with constant and variable h; we select h based on kernel density estimation

Table 17: Performance of ASKIT on datasets with different low dimensional manifolds. Here "$\boldsymbol{m}$" is points per leaf, "$\boldsymbol{s}$" is number of skeleton points, "$\boldsymbol{h}$" the Gaussian kernel bandwidth, "**hit**" is the estimated percentage of correct neighbors (we use $\kappa = 2m$). The relative overall error "$\boldsymbol{\epsilon}$" is estimated by comparing the treecode with direct evaluation on 10K randomly selected points; "$\boldsymbol{\epsilon}_\kappa$" indicates the error if we only use the nearest neighbors (equivalent to a sparse approximation). We report several timings: "$\boldsymbol{T}_\kappa$" is the time to construct nearest neighbor lists and "$\boldsymbol{T}$" is the overall time of the treecode," $\boldsymbol{T}_S$" is the skeletonization time, "$\boldsymbol{T}_E$" is the evaluation time. To illustrate our complexity estimate (24), we report the number of tree nodes visited per point during evaluation. "**near**" is the average number of leaves visited as a percentage of the worst case κ; "**far**" the number of nodes whose skeleton was used to evaluate the far field (at a point) as a percentage of the worst case $\kappa \log(N/m)$. Also "$\boldsymbol{T}_{\text{dir}}$" is the time for a direct N^2 evaluation (estimated using the on $10K$ points). All times are in seconds. We highlight the most important columns: the error, the treecode time, the evaluation time, and also the effectiveness of pruning. "**run**" indexes the experiments and we use it to discuss the results.

	parameters			*errors*			*timings (secs)*				*pruning*	
run	$\boldsymbol{m}$	$\boldsymbol{s}$	$\boldsymbol{h}$	**hit**	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}_\kappa$	$\boldsymbol{T}_\kappa$	$\boldsymbol{T}$	$\boldsymbol{T}_S$	$\boldsymbol{T}_E$	**near**	**far**
	4D Normal distribution, $N = 100,000$,			$T_{\text{direct}} = 16$ *secs*								
1	64	4	0.21	97	5E-02	3E-01	5	5	1	4	15%	3 %
2	64	32	0.21	97	1E-02	3E-01	5	7	<1	6	15%	3 %
3	256	128	0.21	87	2E-03	8E-02	7	8	1	7	5 %	1 %
4	64	4	1.00	97	7E-01	1E+00	5	4	<1	4	15%	3 %
5	64	32	1.00	97	6E-02	1E+00	5	7	<1	6	15%	3 %
6	256	128	1.00	88	3E-03	9E-01	7	8	1	7	5 %	1 %
7	64	4	5.00	97	8E-01	1E+00	5	5	<1	4	15%	3 %
8	64	32	5.00	97	1E-04	1E+00	5	7	<1	6	15%	3 %
9	256	128	5.00	88	4E-08	1E+00	7	8	1	7	5 %	1 %
	4D Normal distribution, $N = 1,000,000$,			$T_{\text{direct}} = 1591$ *secs*								
10	64	4	0.16	95	1E-01	5E-01	56	37	1	36	14%	3 %
11	64	32	0.16	95	3E-02	5E-01	54	71	2	69	14%	3 %
12	256	32	0.16	84	7E-03	3E-01	60	81	3	78	5 %	1 %
	Inexact neighbor information											
13	64	4	0.16	26	3E-01	8E-01	31	63	2	61	25%	4 %
14	64	32	0.16	26	5E-02	8E-01	31	119	3	117	25%	4 %
15	64	128	0.16	27	3E-02	8E-01	31	262	4	257	25%	4 %
	16D Normal distribution, $N = 1,000,000$,			$T_{\text{direct}} = 1630$ *secs*								
16	64	32	0.45	35	1E-03	3E-03	57	573	6	568	83%	23%
17	64	128	0.45	34	7E-04	4E-03	57	725	7	717	83%	23%
18	256	128	0.45	26	1E-04	2E-03	63	997	10	987	55%	8 %
19	64	32	1.76	35	7E-02	1E+00	57	564	6	559	83%	23%
20	64	128	1.76	35	5E-02	1E+00	57	729	7	723	83%	23%
21	256	128	1.76	25	2E-02	1E+00	63	1011	11	1000	55%	8 %
	64D Normal distribution, $N = 1,000,000$,			$T_{\text{direct}} = 1829$ *secs*								
22	64	128	0.75	8	3E-15	3E-15	63	984	10	974	99%	40%
23	64	128	2.62	8	2E-01	1E+00	63	985	11	975	99%	40%
24	64	128	4.98	8	8E-03	1E+00	63	991	10	981	99%	40%

Table 18: Performance of ASKIT on some large datasets. The two last runs (33-34) are done using 16 and 256 Intel sockets respectively and the MPI library [62].

	parameters			*errors*			*timings (secs)*				*pruning*	
run	m	s	h	**hit**	ϵ	ϵ_κ	T_κ	T	T_S	T_E	**near**	**far**
	4D Normal distribution, variable h, $N = 1,000,000$, $T_{\text{direct}} = 1622$ *secs*											
25	64	32	1.00	95	3E-02	1E+00	55	145	2	143	14%	3 %
26	64	32	5.00	95	4E-06	1E+00	57	144	2	142	14%	3 %
	1000D (4d-intrinsic), $N = 1,000,000$, $T_{\text{direct}} = 7315$ *secs*											
27	64	32	0.75	99	6E-02	1E+00	204	393	11	382	14%	3 %
28	64	32	3.75	99	2E-06	1E+00	195	391	10	381	14%	3 %
	18D (UCI SUSY), $N = 5,000,000$, $T_{\text{direct}} = 41000$ *secs*											
29	64	128	0.10	91	6E-09	5E-06	703	1516	34	1482	43%	9 %
30	64	128	0.40	91	1E-01	7E-01	709	1510	32	1478	43%	9 %
31	64	128	1.00	90	7E-02	1E+00	567	1530	33	1497	43%	9 %
32	64	128	5.00	90	1E-03	1E+00	577	1566	33	1533	43%	9 %
	MPI-parallel 128D (4d-intrinsic), $N = 5,000,000$, $T_{\text{direct}} = 280,000$ *secs*											
33	64	128	0.50	100	1E-03	1E+00	233	1339	12	1158	17%	100%
34	64	128	0.50	100	1E-03	1E+00	43	170	1	144	17%	100%

theory [121](Eq. 4.14). Note, that in large d, for certain values of h, K does not have decaying singular values [108, 64]. We include such cases. Also we choose h so that far-field is necessary for accurate summation.

Discussion: For low-accuracy approximation our scheme outperforms the direct evaluation at about 1M points. Depending on the kernel and the accuracy the speed-up can be less dramatic or the cutoff may be at much higher N.

In runs 1–9 we show how the method converges for different bandwidths and m, s values for 100K points that are normally distributed in 4D. Note that the error ϵ converges with increasing m and s. The convergence can be quite rapid (runs 7–9). In most of the runs, the far-field is critical in getting accuracy. To show this, we report ϵ_κ, the error in u constructed using only the $\kappa = 2m$ nearest neighbors for each point. We see that the far field is essential, and truncation does not get a single digit correct. On the other hand for run 22, with $h = 0.75$ the far field is wasted effort. In runs 23–24 the far field is essential. In runs 27-28, we consider a problem in 1000D. The scheme converges quickly and it is

20× faster than the direct evaluation. For the UCI dataset (29-32) ASKIT is 25× faster than the direct evaluation. Runs (1-32) took place on a single node. In runs (33-34) we show a distributed memory run for 5M points in 128D on 16 and 256 nodes resulting a 2000× speed-up over one-socket direct evaluation.

To demonstrate the effects of using the nearest neighbors compare runs 10–12 to runs 13–15. The only difference is that that we use a very approximate search so the hit-rate "**hr**" (correct neighbors/κ) small. As a result the errors are higher and the pruning is not as effective (we visit more leaves and more internal nodes). T_E is almost 3× larger. Finally, notice that we increase the dimension the neighbors in general become less accurate. This is because we use a fixed number of iterations in our greedy neighbor search. The skeletonization costs are negligible compare to the evaluation costs.

3.4 Conclusions

We presented a new scheme for high dimensional N-body problems and conducted a proof-of-concept experimental study. Our scheme is based only on kernel evaluations, uses neighbor-based pruning, and uses neighbor-sampled interpolative decomposition to approximate the far field. Since this method is new, there are many open problems and several opportunities for optimization. The most pressing one is deriving a rigorous error bound that incorporates our sampling scheme; this is ongoing work. There is also further work to be done in optimizing the performance of the scheme, in adaptive determination of the skeleton size, and in improving the sampling. Finally, notice that if we are given similarities and a hierarchical clustering, our scheme does not involve any distance calculations so it should be possible to apply to points (objects) in non-metric spaces.

CHAPTER IV

A HIGH DIMENSIONAL LIKELIHOOD FUNCTION FOR VARIATIONAL IMAGE SEGMENTATION

We consider a supervised image segmentation algorithm using a Bayesian framework and a variational formulation for a maximum a posteriori (MAP) estimation of the label field. In the Bayesian framework, we must define the likelihood function (the probability of the image intensities given the label field) and the prior probability (for the label field). In this chapter, we focus on the likelihood. Typically, the likelihood of the intensity is decoupled for every pixel and is given by a parametric density (often a Gaussian function). Instead, in §4.2, we propose a non-parametric, high-dimensional, kernel density estimation (KDE) for the likelihood function, based on Gabor features (300 per pixel). This approach better approximates non-local correlations. §4.4 provides experimental evidence that this likelihood function performs very well and converges to the correct segmentation with the number of training images.

4.1 Introduction

The main goal of our work is to propose a likelihood function that incorporates non-local information and problem-specific information and give experimental evidence that it can lead to more accurate segmentations than pixel-wise intensity likelihood functions. The framework of our approach is based on interpretation of segmentation as a statistical inverse problem for the label field [33]. Such an approach is well known and has been used in many contexts in computer vision, for example image retrieval, shape recognition, and motion estimation. Here, we limit our discussion on grayscale images (our target application is medical imaging), for which large training samples may exist.

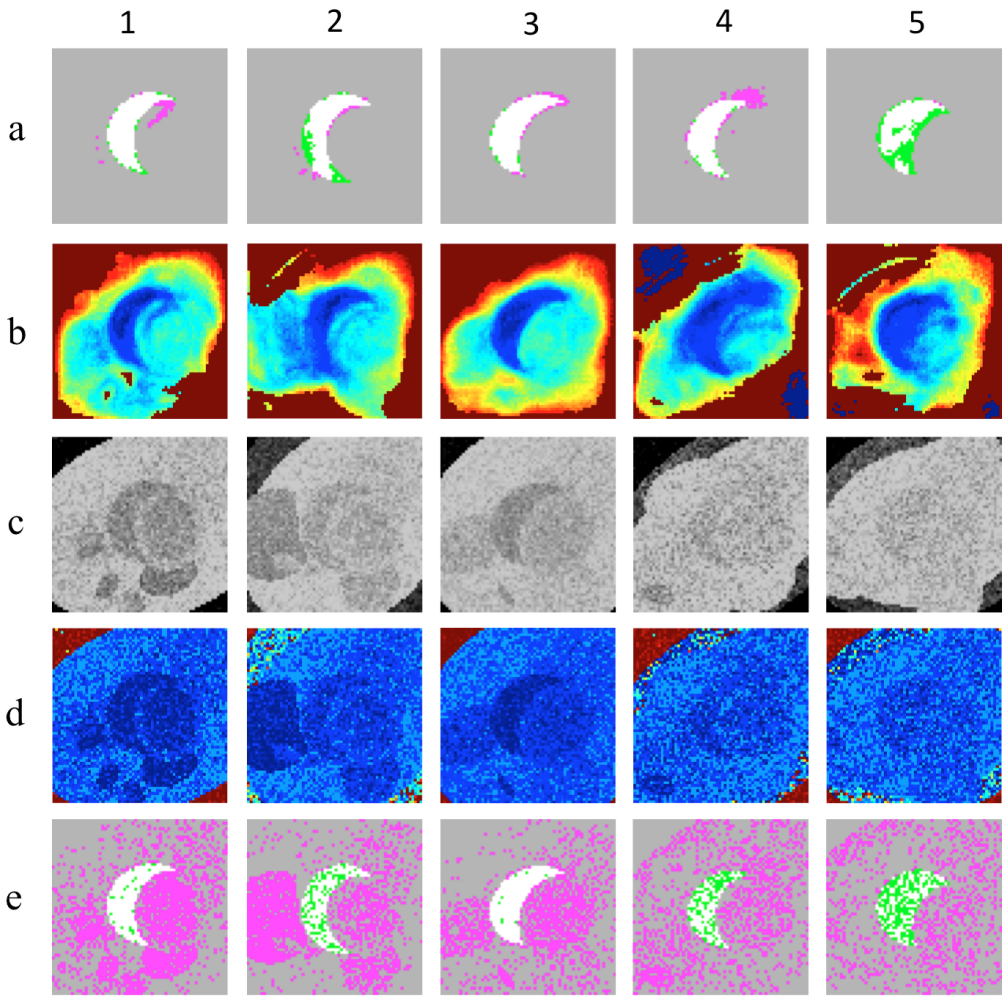


Figure 25: *Illustration of the performance of our scheme*: we demonstrate the effect of the likelihood function to the quality of a binary segmentation (to "object" and "background") using a Bayesian variational framework. The **row (c)** shows the testing images (created synthetically) we have used to test our algorithm. The **row (b)** depicts the likelihood function using Gabor features and a high-dimensional KDE method; blue indicates a higher probability that a pixel belongs to the "object" class. The **row (a)** depicts the segmentations using the Gabor-KDE likelihood. The **row (d)** depicts the likelihood function using a pixel-wise intensity computed with one-dimensional intensity-KDE scheme. The **row (e)** gives the resulting segmentation after using the Laplacian smoothness prior. To indicate the difference between our segmentations of the true segmentations, we use '**white**' indicates the overlap region between the true label and our segmentation, '**green**' indicates the left region of the true label excluding the overlap, and '**magenta**' represents the remaining region of our segmentation besides the overlap.

4.1.1 Our methodology and contribution

We consider only the case of binary segmentation in this chapter. For the multi class case, it is easy to convert to several binary segmentation subproblems. Given an image $\mathbf{I} \in \mathbb{R}^N$, where N is the number of pixels, we seek to find a label field $\boldsymbol{\phi} \in \{0,1\}^N$ find the label field $\boldsymbol{\ell}$ as the MAP point of $p(\boldsymbol{\phi}|\mathbf{I}) \sim p(\mathbf{I}|\boldsymbol{\phi})p(\boldsymbol{\phi})$, the first term being the likelihood of $\mathbf{I}$ and the second the prior of $\boldsymbol{\phi}$. The prior is defined using a standard smoothness prior which corresponds to a generic Gibbs-type probability. For the likelihood $p(\mathbf{I}|\boldsymbol{\phi})$, we first compute Gabor features for every pixel and then we assume that the Gabor features of each pixel conditioned on the label of the pixel are independently and identically distributed (i.i.d). To estimate this distribution, we use our own non-parametric nearest-neighbor KDE. To find the MAP estimation, we relax the integrality condition for $\boldsymbol{\phi}$ and we only require that $\boldsymbol{\phi} \in [0,1]^N$, and we solve an inequality-constrained quadratic program to compute $\boldsymbol{\phi}$.

In standard practice, the likelihood $p(\mathbf{I}|\boldsymbol{\phi})$ is computed by assuming that the pixel intensities conditioned to the label are i.i.e. This assumption is motivated by pragmatic reasons. For example in medical imaging [], in the absence of large-enough training sets that's the best one can do. Also the computational challenge of working with high-dimensional KDEs is considered prohibitive. Our main contribution is to demonstrate that this assumption is problematic, propose a framework for alternative representations, leverage our work on fast algorithms for high-dimensional KDE, and to provide preliminary experimental evidence this new framework performs quite well: it is robust to noise, allows full use of training images (and not only some average), does not require any parameter selection, requires no initialization and is general and extensible (e.g., it be readily coupled with more sophisticated priors like total variation or Mumford-Shah). In Figure 25, we demonstrate the performance of our methodology for synthetic images that have low contrast, low resolution, and highly noisy regions.

4.1.2 Limitations

The assumption that the pixel-wise values of the features are i.i.d, is in correct. This assumption can also be relaxed but this is beyond the scope of this paper. Our focus is the formulation of likelihood, and we have selected the simplest possible prior, a Laplacian smoothing. This prior makes the solution of the labels easier computationally but is by no means an essential part of our formulation. Also, the feature selection is rather standard and we do not discuss possible alternatives both with regards the features but also the distance metric in the feature space. The Gabor features are rather sensitive to scaling transformations of the label field. Also, we do not discuss the case of corrupted or inexact training data. The requirement for a large set of training images potentially limits the applicability of the proposed likelihood method.

4.1.3 Related work

Variational approaches is a main research field in medical image segmentation. Various variational methods try to minimize a functional of ϕ by designing appropriate functionals that generated connected regions while enforcing boundary smoothness constraints. Examples of such methods include active contour [74], level set [98, 17], geometric active contour [16], and minimization of the Mumford-Shah functional [94].

The majority of the methods, with the exception of supervised learning methods, discussed above do not make extensive use of prior information. But if such information is available, typically in the form of large training sets, we should consider methods that use it. Such a method is Bayesian inference. A lot of the methods we discussed can in fact be viewed as variational methods to compute a MAP point. The statistical interpretation of variational image segmentation is nicely summarized in [33]. In [34], the authors use a similar formulation, where the likelihood is taken as a pixel-wise intensity with a normal distribution, the prior is a smoothness prior (weighted total variation). Additional methodologies using one-dimensional KDE estimation either for the likelihood or the prior are

discussed in [34, 6]. In [21], an image registration technique is used to define the priors, but using only with rigid deformations, which limits the applicability of this method significantly.

The statistical interpretation of variational image segmentation is nicely summarized in see [33]. In [34], the authors use a similar formulation, where the likelihood is pixel-wise intensity with a normal distribution, and the prior is a smoothness prior (weighted total variation) and quadratic shape prior constructed under the assumption that the distribution of shapes is normally distributed. Additional methodologies on one-dimensional KDE estimation either for the likelihood all the prior are discussed in [34, 6]. In [21], registration techniques are used to define the priors and works well with rigid priors, but these are not applicable to our case since we consider shapes that have non-rigid variability. Example of intensity-based likelihood functions and most sophisticated priors in the context of medical imaging can be found in [111] Examples of alternative likelihood functions, include the piecewise constant Mumford-Shah functional use the mean color value to represent the color in each region, and aims to minimize the color variance in each region[94, 17].

4.2 Problem Formulation

As we mentioned in the introduction $\mathbf{I}$ and $\boldsymbol{\phi}$ indicate the discrete grayscale image and label values at the pixels. Given the posterior distribution $p(\boldsymbol{\phi}|\mathbf{I}) \sim p(\mathbf{I}|\boldsymbol{\phi})p(\boldsymbol{\phi})$, the MAP estimate is found by maximizing the log-likelihood:

$$\boldsymbol{\phi}^* = \text{argmax}_{\boldsymbol{\phi}} \log p(\mathbf{I}|\boldsymbol{\phi}) + \log p(\boldsymbol{\phi}) = E_l + E_p, \tag{26}$$

where E_l is the likelihood term and E_p is the prior term. Typically $p(\boldsymbol{\phi})$ is given as a Gibbs distribution $\sim e^{-G(\boldsymbol{\phi})}$. Then we can set $E_p = -G(\boldsymbol{\phi})$ since the proportionality constant is not needed in the computation of $\boldsymbol{\phi}^*$.

In the literature, a typical assumption for the likelihood function is that the conditional

intensity is i.i.d., that is

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{j=1}^{N} p(\mathbf{I}_j|\boldsymbol{\phi}_j). \tag{27}$$

Typically, a parametric density is assumed for $p(\mathbf{I}_j|\boldsymbol{\phi}_j)$ and the parameters are estimated using the training set.

We take a different approach. Let us first define the feature map $\mathbf{G} : \mathrm{R}^N \rightarrow \mathrm{R}^{d\times N}$ that maps scalar intensities to a $d-$dimensional vectors so that $\mathbf{g}_j = \mathbf{G}(I)_j \in \mathcal{R}^d$ is the feature vector of pixel j. To define the likelihood, we assume that *the features* are conditionally independent so that

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{j=1}^{N} p(\mathbf{g}_j|\boldsymbol{\phi}_j) = \prod_{j=1}^{N} p(\mathbf{G}(\mathbf{I})_j|\boldsymbol{\phi}_j). \tag{28}$$

(This equation is also an approximation because the features are correlated.)

For a binary segmentation to "*object*" ($\boldsymbol{\phi}_j = 1$) and "*background*" ($\boldsymbol{\phi}_j = 0$), equation (28) simplifies to

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{j:\boldsymbol{\phi}_j=1} p^{obj}(j) \prod_{j:\boldsymbol{\phi}_j=0} p^{bg}(j) \tag{29}$$

where $p^{obj}(j) := p(\mathbf{g}_j|1)$ and $p^{bg}(j) := p(\mathbf{g}_j|0)$. With this notation, the log-likelihood can be written as $E_l = \sum_{j=1}^{N} \boldsymbol{\phi}_j \log p^{obj}(j) + (1 - \boldsymbol{\phi}_j) \log p^{bg}(j)$.

In the infinite dimensional setting (which is convenient for defining the prior), the above log-likelihood for an image function $I(x)$ and a label function $\phi(x)$, becomes

$$p(\mathbf{I}|\boldsymbol{\phi}) = \int_x \phi(x) \log p^{obj}(x) + (1 - \phi(x)) \log p^{bg}(x) \tag{30}$$

With simple algebraic manipulations, this equation becomes $E_l = \int_x \phi(x) \log \frac{p^{obj}(x)}{p^{bg}(x)}$ (plus a constant). For the discretized case, $E_l = \mathbf{b} \cdot \boldsymbol{\phi}$, where

$$\mathbf{b}_j = \log \frac{p^{obj}(j)}{p^{bg}(j)}. \tag{31}$$

Given this definition of likelihood, we need to decide on the approximation of p^{obj} and p^{bg} given the training samples. We use a kernel density estimation approach in which we

compute the features of the training images and we compute the density for each class. We discuss this and related approaches in detail in §4.2.2.

Prior distribution: To complete the MAP estimation we need to define the prior distribution for $\boldsymbol{\phi}$. The training-set can be used to construct a priors [37]. For this time being, we use a simple smoothness prior to isolate the effects of the likelihood function selection. In the infinite-dimensional setting, we define the prior by setting $G(\phi) = \frac{\beta}{2}\int_x \nabla\phi\cdot\nabla\phi$, which upon discretization becomes $G(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{\phi}\cdot\mathbf{L}\boldsymbol{\phi}$, where $\mathbf{L}_\beta$ is the discrete Laplacian weighted by $1/\beta$, a regularization parameter that is related on the level of noise. We use a simple five-point-stencil Laplacian with Neumann conditions do define the Laplacian operator. This constraint does not enforce integrality. There are many ways to enforce it, for example using phase fields or level sets. We simply add an inequality constraint $0 \leq \boldsymbol{\phi}_j \leq 1$.

In summary, our formulation consists of solving the following problem

$$\min_{\boldsymbol{\phi}_j\in[0,1]} \frac{1}{2}\boldsymbol{\phi}\cdot\mathbf{L}_\beta\boldsymbol{\phi} + \mathbf{b}\cdot\boldsymbol{\phi}, \tag{32}$$

where $\mathbf{b}_j$ is given by (31). This optimization is a convex quadratic program that can be easily solved at least for modest (two dimensional) image resolutions. The regularization parameter β can be computed using cross validation and the training set [126]. Next we describe the KDE scheme we use to compute $\mathbf{b}$.

4.2.1 Gabor Features

In Eqn. (51) we assume the features are independent. Unfortunately features at different pixels, especially the nearby pixels, are usually correlated. In order to contain enough spatial information at each pixel, we choose the frequently used Gabor wavelet features to sample spatial information and correlation at each pixel [115, 133, 45].

In general, a Gabor function in space domain can be treated as a product of a 2D-Gaussian-shaped function, known as the envelope, and a sinusoid, know as the carrier. The Gaussian envelop could have different scales and rotations to emphasize different details. The frequency in the carrier function determines the details in the gaussian envelop. The

higher frequencies always filter out the information of edges, lines, and even noise, and the lower frequencies tend to average the content in a Gaussian window. By this way, Gabor filters with different scales, orientations and frequencies could represent images in different resolution. Gabor representation is optimal in the sense of minimizing the joint two-dimensional uncertainty in space and frequency [41]. Since Gabor features are constructed by filtering an image along different orientations with different window sizes, they contain information related to correlation of the pixel intensity. As shown in Figure 26, with different window sizes, orientations and frequencies, Gabor features characterize different details of the same input image.

The 2-dimensional Gabor function $g(x, y)$ can be written as

$$h(x,y) = \frac{1}{2\pi\sigma_x\sigma_y}\exp\left\{-\frac{1}{2}\left(\frac{x^2}{\sigma_x^2}+\frac{y^2}{\sigma_y^2}\right)\right\}\exp(2i\pi fx) \tag{33}$$

where σ_x and σ_y is the height and width (scale) of the 2d Gabor filter size, f is the frequency of the sinusoid wave. Gabor functions form a complete but nonorthogonal frame set. Expanding a signal using this frame gives a localized frequency description. Let $h(x, y)$ be the mother Gabor wavelet, the the dictionary can be obtained by dilations and rotations in the form of

$$h_{mn}(x,y) = a^{-m}g(x\cos\theta + y\sin\theta, -x\sin\theta + y\cos\theta) \tag{34}$$

where $a > 1, m, n \in \mathcal{N}, \theta = n\pi/n_a$, n_a is the total number of orientations. In all of our experiments, we use 4 scales $\sigma_x = \sigma_y = 4, 8, 16, 32$, and $n_a = 8$. The frequency is adaptively chosen according to different scales. Given the 2D Gabor functions, the Gabor wavelets can be obtained by passing the input image through those Gabor filters,

$$I[x,y] \otimes h[x,y] = \sum_{n_1=-\infty}^{\infty}\sum_{n_2=-\infty}^{\infty} I[n_1,n_2]h[x-n_1, y-n_2] \tag{35}$$

If we extend the 2D image and Gabor filter to a long vector by concatenating each rows, for every position (x, y) (index i), the feature map $\mathbf{g}_i = \mathbf{G}(\mathbf{I})_i$ in Eqn. (52) can be written as

$$\mathbf{g}_i = \mathbf{G}(\mathbf{I})_i = \mathbf{G}_i\mathbf{I} \tag{36}$$

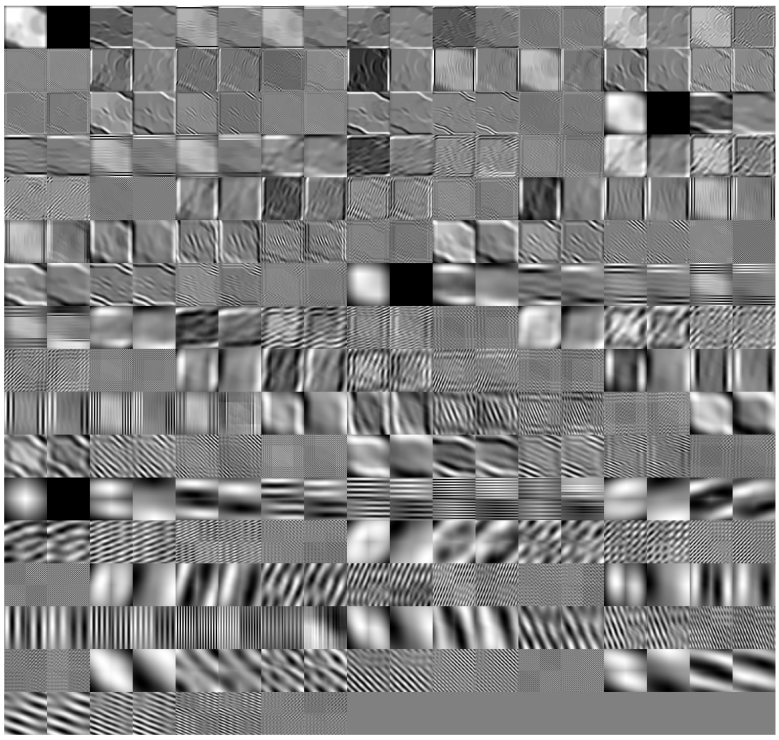


Figure 26: *Illustration of Gabor features:* This first cell shows the input image, and the following images illustrate the real part and imaginary part of Gabor features with different parameters ($\sigma_x = \sigma_y = 4, 8, 16, 32, n_a = 8$). In order to display different Gabor features, every dimension of the Gabor features have been rescaled to the range $[-1, 1]$.

where $\mathbf{G}_i =\in \mathcal{R}^{d\times N}, \mathbf{I} \in \mathcal{R}^N$. Each row of $\mathbf{G}_i$ corresponds to one Gabor filter with a specific orientation, scaling and frequency. The element in d-th row and t-th column $\mathbf{G}_{i,(d,t)}$ is the corresponding values of the $h[x-n_1, y-n_2]$, where index i refers to a position (x, y); index t refers to (n_1, n_2) and index d refers to the d-th Gabor filter.

4.2.2 High Dimensional Kernel Density Estimation

There have been many approaches for approximating the likelihood function, but typically are limited either in very low dimensional KDE estimation methods, parametric approaches (with a small number of parameters), or mixtures of Gaussians. Examples include

histogram approximations [112, 34], Gaussian approximations [21], or mixture of Gaussians [21, 23]. Parametric approaches and Gaussianity assumptions (or mixtures with a fixed size) are not always well-justified and if the true distribution is not consistent with those assumptions, their performance will not be improved with increasing training sample size.

Non-parametric methods on the other hand, are more descriptive but involve significant computational costs, especially in high dimensions. KDE, under mild assumptions, was shown to converge to the true distribution in the limit of infinite sample size [120] and the sample size depends on the **intrinsic dimensionality** rather than the ambient dimensionality of the underlying distribution [103, 100]. Given the effectiveness of KDE methods and the evidence of low dimensional structure of feature space, we opted for a KDE approach based on Gabor wavelet features.

In more detail, denote $\mathbf{g} \in \mathcal{R}^d$ as a point in d-dimensional feature space, $\{\mathbf{a}_1, \cdots, \mathbf{a}_k, \mathbf{a}_i \in \mathcal{R}^d\}$ are data points sorted in an order of increasing distance between $\mathbf{a}_i$ and $\mathbf{g}$, i.e., $\|\mathbf{g} - \mathbf{a}_1\| \leq \|\mathbf{g} - \mathbf{a}_2\| \leq ... \leq \|\mathbf{g} - \mathbf{a}_k\|$.

The balloon kernel density estimator is defined as

$$p(\mathbf{g}_j|\boldsymbol{\phi}_j = c) = \frac{1}{n_c} \sum_{\boldsymbol{\phi}_i = c} K(\mathbf{g}_j, \mathbf{g}_i) \tag{37}$$

where n_c is number of points from class c. We use the Gaussian kernel,

$$K(\mathbf{g}_j, \mathbf{g}_i) = (2\pi)^{-d/2} \rho_j^{-d} \exp\left(\frac{\|\mathbf{g}_j - \mathbf{g}_i\|^2}{2\rho_j^2}\right). \tag{38}$$

where ρ_j is the bandwidth for $\mathbf{g}_j$, which is defined $\rho_j = \|\mathbf{g}_j - \mathbf{g}_{k_\rho}\|$, the distance between $\mathbf{g}_j$ and its k_ρth-nearest neighbor in the training dataset.

There are two main issues in balloon KDE: how to choose the proper value of k_ρ and how to calculate the kernel summation quickly. The naive k-nearest neighbors search and the naive kernel summation both require $\mathcal{O}(n^2)$ work. If n is large, it is impractical to evaluate them directly. We will discuss how to find the k nearest neighbors fast in §**??**.

Once the nearest neighbors have been computed, we can select the appropriate bandwidth. However, unless the bandwidth is small, the Gaussian kernel cannot be truncated. There exist fast approximate summation methods that do not truncate the kernel but instead approximate it when $\|\mathbf{g}_j - \mathbf{g}_i\|$ is large. Since we are only interested in the effectiveness of the segmentation and not on the actual probability density, we simply summarize only the first k_t-nearest points. That is

$$p(\mathbf{g}_j|\boldsymbol{\phi}_j = c) \approx \frac{1}{n_c} \sum_{i=1,\boldsymbol{\phi}_i=c}^{k_t} K(\mathbf{g}_j, \mathbf{g}_i) \tag{39}$$

Therefore we need to determine two parameters: k_ρ where $\rho_j = \|\mathbf{g}_j - \mathbf{g}_{k_\rho}\|$, and k_t in Eq. 39 to truncate the summation. For a given training set, k_ρ and k_t can be selected by cross validation (CV). The CV procedure is given in Algorithm 18. The evaluation function $J(k_\rho, k_t)$ is defined as the segmentation error using the likelihood function in Eq. 39. In more details, let $\boldsymbol{\phi}_j^s$ be the estimated label of pixel j, $\boldsymbol{\phi}_j^t$ is its true label, we have

$$\boldsymbol{\phi}_j^s = \begin{cases} 1, & \text{if } p(\mathbf{g}_j|\boldsymbol{\phi}_j = 1) > p(\mathbf{g}_j|\boldsymbol{\phi}_j = 0) \\ 0, & \text{if } p(\mathbf{g}_j|\boldsymbol{\phi}_j = 1) < p(\mathbf{g}_j|\boldsymbol{\phi}_j = 0) \end{cases} \tag{40}$$

$$J(k_\rho, k_t) = \frac{ones(\boldsymbol{\phi}^s \text{ and } \boldsymbol{\phi}^t)}{ones(\boldsymbol{\phi}^s \text{ or } \boldsymbol{\phi}^t)} \tag{41}$$

where $J(k_\rho, k_t)$ is defined as the Jaccard index of the segmentation for object. Since each element of $\boldsymbol{\phi}$ is either 0 or 1, the intersection of conjunction can be obtained easily by bitwise 'or' and 'and' operation. '$ones(\cdot)$' counts the number of ones in the input vector.

Complexity analysis: Suppose finding $\mathbf{g}_j^{val}$'s k_t nearest neighbors among the training data takes $\mathcal{O}(\alpha)$ operations. For each $\mathbf{g}_j^{val}$, we need to evaluate $p(\mathbf{g}_j^{val}|\boldsymbol{\phi}_j^{val} = c)$ in Eqn. 39, it takes $\mathcal{O}(k_t)$. We need to look from 1 to k_t to find the best k_ρ^*, hence in order to find the best k_ρ^*, it takes $\mathcal{O}(k_t^2)$ operations for every validating data. In a cross validation scheme, each data point is used once as validating points, hence the total operation to find k_ρ^* with each k_t takes $\mathcal{O}(k_t^2)$. There are at most $\log k_{t,max}$ possible k_t, hence the total time complexity is $\mathcal{O}\left((\alpha + nk_t^2)\log k_{t,max}\right)$. If training data points are grouped in tree structure, k_t near

Algorithm 18 CV

split all $\mathbf{g}_i$ into V groups
for $v = 1 : V$ **do**
  tack vth group as validating set $\{\mathbf{g}_j^{val}\}$, the remaining as training set $\{\mathbf{g}_i^{trn}\}$
  initial k_t, ξ to make the CV start
  find k_t near neighbors for every $\mathbf{g}_j^{val}$ from $\{\mathbf{g}_i^{trn}\}$
  find $k_\rho^* = \min_{k_\rho} J(k_\rho, k_t), k_\rho = 1, 2, \cdots, k_t$
  $J_{old} = J(k_\rho^*, k_t)$
  while $\xi > \eta \ \ and \ \ k_t < k_{t,max}$ **do**
    $k_t = 2k_t$
    find k_t near neighbors for every $\mathbf{g}_j^{val}$ from $\{\mathbf{g}_i^{trn}\}$
    find $k_\rho^* = \min_k J(k_\rho, k_t)$
    $\xi = |J(k_\rho^*, k_t) - J_{old}|/J_{old}$
    $J_{old} = J(k_\rho^*, k_t)$
  end while
end for
return k_ρ^*, k_t

neighbors can be found in time $\mathcal{O}(n \log n + nk_t)$. The details is given in §2.3.4. As a result, the total time complexity of this CV procedure is $\mathcal{O}((n \log n + nk_t + nk_t^2) \log k_{t,max})$.

4.3 Nearest neighbors and low dimensional structures

The cost of naive nearest-neighbor computation is $\mathcal{O}(n^2)$, where n is the number of images in the training set times the number of pixels per image. As n increases, it will become prohibitively expensive to directly search for the nearest neighbors. We have developed methodology and algorithms for high-dimensional KDE. In our experiments, we tested two approaches to find the neighbors, an exact metric tree, another is random projection kd-tree to compute approximate neighbors. For the details of metric tree and random projection tree, please refer to []. Using 1000 training images of size 256×256, for pixels of one single testing images, finding the 1000 nearest neighbors for every pixel in the 176 dimensional Gabor feature space took dozens of seconds in parallel on 64 x86 cores in our experiments.

Many applications in face recognition have shown that Gabor features have redundancy and low dimensional structure [41, 85]. People have demonstrated that even with high ambient dimensionality, it is still possible to obtain good efficiency for nearest neighbors

search if data lie along a low dimensional sub-manifold [72, 107, 11]. That is a good hint that even the approximation tree approach still give promising results.

Table 19 shows the pruning percentage of the exact metric tree search. The basic idea of pruning is when we search for the nearest neighbors of a certain point, only part of the reference points need to be looked at. Thus the pruning percentage indicates how many points can be discarded when we find the nearest neighbors. The pruning percentage on a single leaf node i is defined as

$$prune\% = \frac{N - n_i}{N - N/m_l} \tag{42}$$

where N is the total number of query points; n_i is the number of query points on a single leaf node i; m_l is the number of leaf nodes. Usually for the metric tree search, when k is large, the searching radius enlarge. As a result, it tends to make the tree pruning lose its efficiency. However, due to the low dimensionality of our dataset, we observed significant pruning.

Table 19: *Pruning Effect of the Exact Metric Tree:* We computed the all nearest neighbors for the points themselves using the Gabor features of each pixel from 100 images. The minimum, maximum, and average percentage of pruning across all leaf nodes on the tree are reported respectively for different k's.

k	2	10	50	100	500
max (%)	99.70	99.48	99.20	99.05	98.68
min (%)	92.33	82.83	74.17	70.80	62.39
avg (%)	95.86	90.39	84.89	82.35	75.23

Figure 27 gives the convergence rate of the random projection tree method, where the miss rate are the percentage of the neighbors found are not the real nearest neighbors. People have demonstrated that the convergence rate of a random projection tree data structure only on the intrinsic dimensionality, rather than the ambient dimensionality of data [39, 52]. Experimentally we observed that eight iterations are enough to find accurate neighbors even for some large $k = 500$.

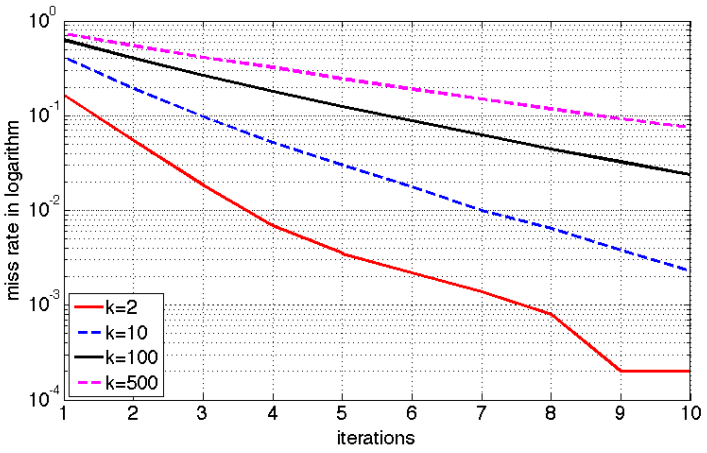


Figure 27: *Miss Rate of the Random Projection KD Tree:* We computed the all nearest neighbors for the points themselves using pixels from 100 images. The error of the random projection tree are given for different k's. It is shown that even with high ambient dimensionality, the tree algorithm still quite efficient to find the true neighbors.

Figure 28 shows the contours of the distance matrix between a given pixel j and all the pixels within one image $\mathbf{g}$ in the feature space. The distance matrix has the form

$$DM(j^{trn}, \mathbf{g}^{tst})[i] = dist(\mathbf{g}_j^{trn}, \mathbf{g}_i^{tst}), \forall i \in \mathbf{g}^{tst} \tag{43}$$

Contours of distance matrix are illustrated for two different points, which are denoted by $'+'$ and $'\times'$. It is easy to see through Gabor features, we could find the similar patterns from other images. For example, the point $'+'$ indicates the upper corner of the object. Ideally its similar points should locate on the upper corners in other images as well. As in the second row in Figure 28 shows, although it is not exactly indicate the the upper corners, the contour of smallest distance do contain the upper corner region. And if the shape is more similar, the distance is smaller, i.e., the smallest distance in image C is smaller than in image A and B. If we consider the intensity only, then the whole region of object in C should be in the same contour, which make us tend to conclude wrong decision.

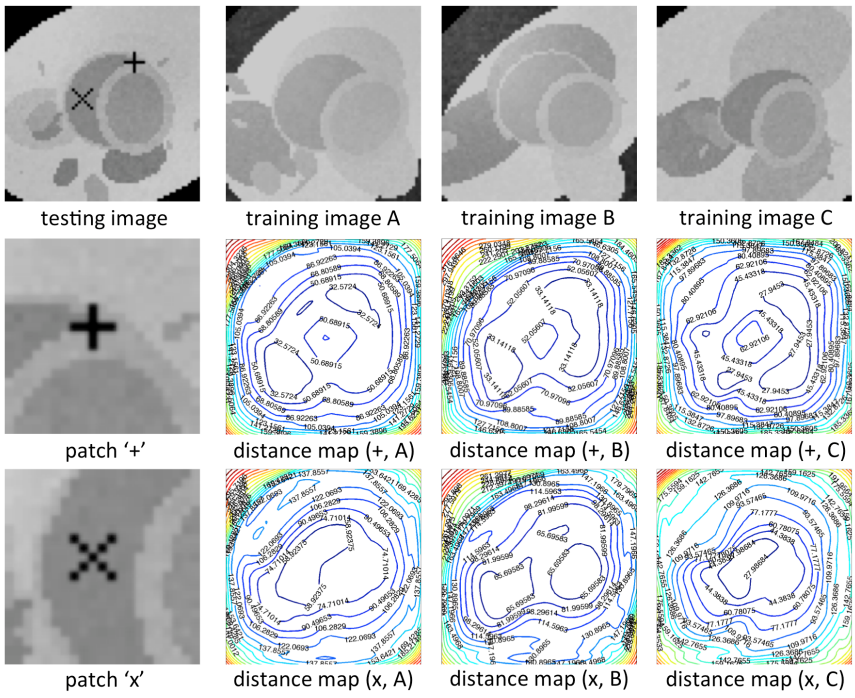


Figure 28: *Illustration of similarity measurement by distance of Gabor features:* The second and the third rows illustrated the distance between two pixels from the testing image and all the pixels from 3 different training images. The color from dark blue to dark red reflects the distance from small to large. And the first images in the second and the third row are the enlarged patch centered at those two pixels.

4.4 Experimental Results

In this section, we firstly describe how our dataset was created, then in section 4.4.2, we describe the accuracy measurement. Section 4.4.3 describes how the log-likelihood is affected by the number of training samples, noise levels, and parameters of balloon KDE. Section 4.4.4 compares the influence of different Gabor filter configurations. In Section 4.4.5, we compare our method with several primary general segmentation approaches. Section 4.4.6 shows the segmentation result with brain MRI. Finally Section 4.4.6.1 compares the likelihood function with SVM.

4.4.1 Dataset used

The performance of segmentation of our proposed approach is evaluated on two datasets: a synthetic one and a brain MRI dataset.

At first we briefly describe the generation of the synthesized data. Figure 29 illustrate the process of synthesizing images. The synthesized images are composed of ellipses with different lengths of short and long axis, rotations, translations. For each ellipse, dozens of random parameters is used to control its shape: the center position x and y, the long axis length a and short axis length b, rotation angle θ; the intensities of each ellipse are also controlled by several parameters: the basic intensity is sampled from a mixture of 3 gaussians with different mean $\mu_i, i = 1, 2, 3$ and standard deviation $\sigma_i, i = 1, 2, 3$. Once we sample the intensity, the ellipse pass one circular average filter with a uniform distributed radius and one motion filter with uniformly distributed length and angle. In summary there are 14 parameters to control the shape and texture of one ellipse. Except for the object region, we add four other partially overlapped ellipses of similar size to the object as large artifacts, and a random number of smaller ellipses as small artifacts. Finally Gaussian noise is added to the whole image.

As a result there are roughly 100 parameters to control the appearance of the whole

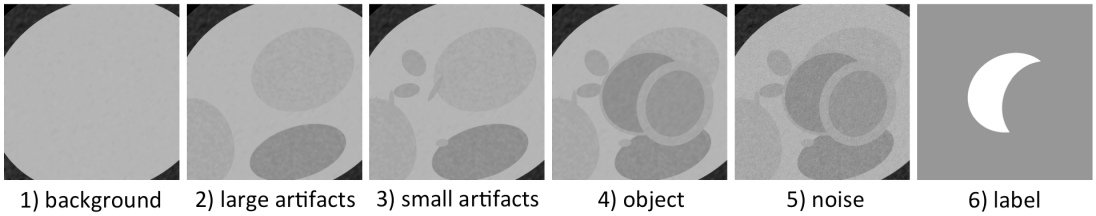


Figure 29: *Generation of synthesized data:* The basic process of generating the artificial data is illustrated. We add both some large artifacts which have similar size of the object and some small artifacts with different intensity. Finally we add some noise on the whole image. The object is some kind of moon shape region as shown in 6).

synthesized images, and for a single pixel, its intensity is determined by dozens of parameters. When we analyze images by Gabor features, the freedom of parameters (intrinsic dimensionality) is obviously lower than the real ambient dimensionality of Gabor features, which in our experiments is 176.

Figure 30 gives some examples of brain images used in our experiments. This figure gives two examples of brains in different slices of a 3d MRI scan. There are three types of data, which we denote them as easy, hard, hardest respectively. They are images under different noise level and blurriness. For the hardest case, even for human eyes, it is difficult to tell the difference between different tissues.

4.4.2 Segmentation Accuracy Measurement

The measure used to evaluate the accuracy of the segmentation in our work is the Jaccard index. Given two sets of position index T and E, the Jaccard index is defined as the ratio of the intersection of T and E over the union of T and E.

$$J(A, B) = \frac{|T \cap E|}{|T \cup E|} \tag{44}$$

where T is the true index of object pixels, and E is the object index given by the segmentation approach. $|T|$ indicates the number of elements in the set T. Ideally if the segmentation is totally correct, then $J(A, B)$ should be 1. The higher the index is, the more accurate the

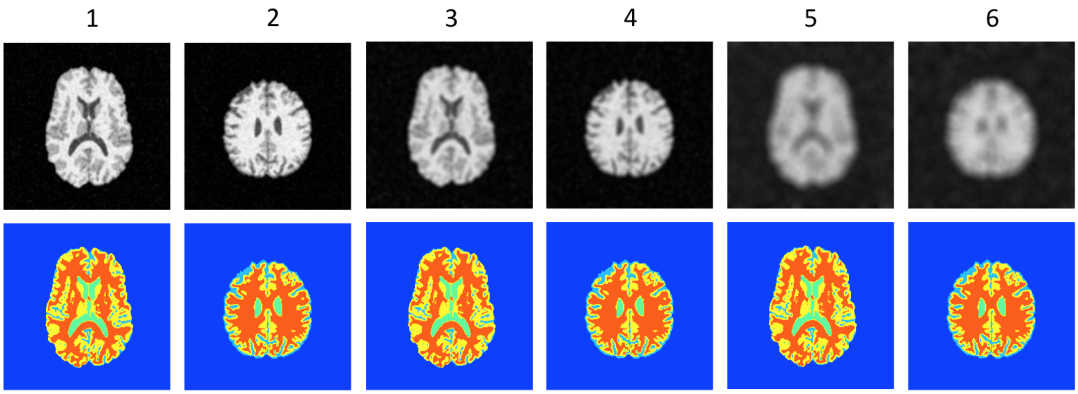


Figure 30: *Examples of brain MRI:* The first row illustrates some brain images we tested in our experiment, and the second row gives their corresponding labels. In this paper, we try to segment the white matter (orange region) and grey matter (yellow region). Figure 1 and 2 are images of different slices, and figure 3 and 4 are corresponding images with more noise and blurriness (hard). Figure 5 and 6 are images with most noise and blurriness (hardest).

segmentation is. More precisely, we use two Jaccard indices,

$$J^{ob} = \frac{|T(ob) \cap E(ob)|}{|T(ob) \cup E(ob)|}, \quad J^{bg} = \frac{|T(bg) \cap E(bg)|}{|T(bg) \cup E(bg)|} \tag{45}$$

4.4.3 The likelihood function

Table 20 compares the performance using the image intensity and the Gabor features. The Jaccard index is based on the segmentation results only considering the likelihood term, i.e., set $\beta = 0$ for the smoothing energy. All results are evaluated as the average Jaccard index of 100 testing images.

At first, compared with intensities, Gabor features capture pixel correlations. That is mainly because the image contains artifacts and noise, and the intensity itself cannot be discriminative enough to identify the correct label. Unlike the intensity, Gabor features is able to characterize the texture patterns and insensitive to noise, which make the labels more separable. We can observe a huge improvement in terms of J^{ob}, which implies the Gabor features give us a much more reliable segmentation of objects. On the other hand, Gabor features is much more robust to noise. The accuracy do not degrade to much even

there exist large noise (20%). Secondly, the representation of the likelihood is convergent with the size of the training sample. Since the KDE does not assume any type of distribution of data, it is more flexible and accurate.

Table 20: *Segmentation accuracy based on KDE*: This table gives the segmentation accuracy using likelihood only ($\beta = 0$). We tested both intensity and Gabor features using 4 scales and 8 orientations. We also tested images under different noise levels from 0% to 20%. It is obvious that the Gabor features outperform over intensity.

noise \ #trn		J^{ob}				J^{bg}			
		1	*10*	*100*	*1000*	*1*	*10*	*100*	*1000*
0%	Intensity	0.2367	0.3633	0.3453	0.3371	0.8039	0.8023	0.7920	0.7809
0%	Gabor	0.5730	0.7863	0.8850	0.9291	0.9386	0.9715	0.9876	0.9930
1%	Intensity	0.2499	0.3288	0.3164	0.3099	0.7494	0.7932	0.7733	0.7622
1%	Gabor	0.5272	0.7588	0.8673	0.9167	0.9376	0.9680	0.9852	0.9916
10%	Intensity	0.1935	0.2391	0.2319	0.2277	0.6169	0.7412	0.7202	0.7086
10%	Gabor	0.5547	0.7525	0.8506	0.8883	0.9352	0.9669	0.9833	0.9885
20%	Intensity	0.1688	0.1993	0.1945	0.1962	0.5787	0.7051	0.6849	0.6774
20%	Gabor	0.5148	0.7316	0.8264	0.8613	0.9331	0.9639	0.9800	0.9853

Figure 31 gives some examples of the log-likelihood $\log \frac{p_{bg}}{p_{ob}}$ calculated by different numbers of training images under different level of noise. We can see the weights themselves are discriminative to tell where are the most likely to be objects or backgrounds. Increasing the training samples would enhance the contrast between two classes. Let us take the second image as an example. There is a darker region on the left side of the target object, and it is very close to the object. If the training images are not enough, it is possible to treat them as points from the same class. In $ntrn = 10$ case, most of the right darker region have a similar log-likelihood as the points from object. With more training data, the log-likelihood of the darker region decrease quickly.

Table 21 compares different likelihood models. In details, we model the probability density function (pdf) of the likelihood as parametric mixture of Gaussians (MoG) and

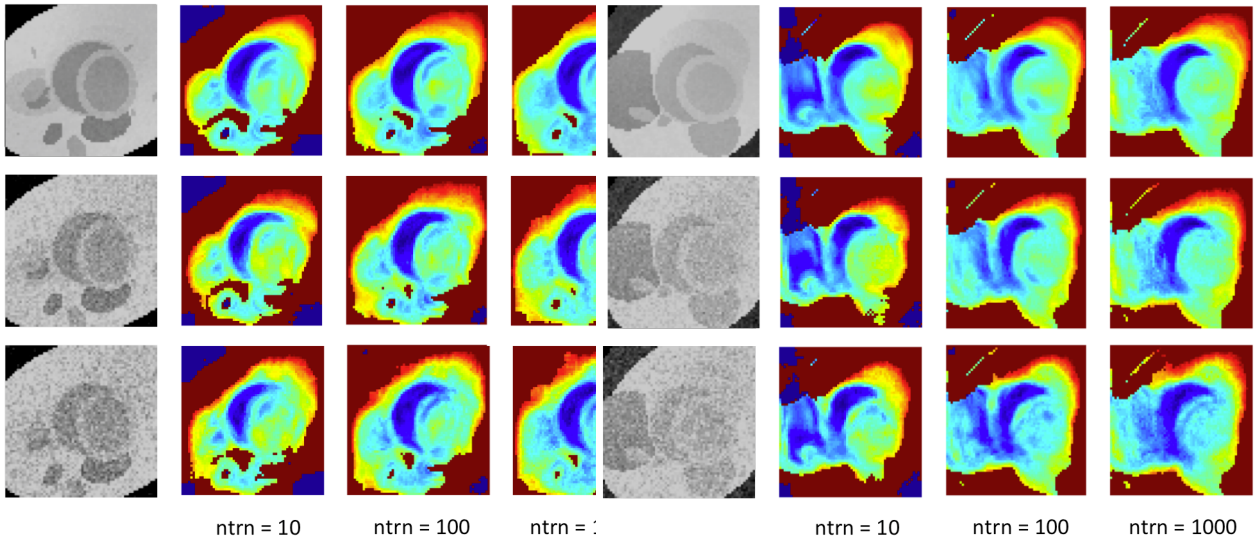


Figure 31: *Examples of log-likelihood* $\log \frac{p_{bg}}{p_{ob}}$*:* The first row gives two clean images, and their corresponding log-likelihood $\log \frac{p_{bg}}{p_{ob}}$ with different number of training images. The second row gives results for the same image with 10% gaussian noise, and the third row shows images with 20% noise.

nonparametric kernel density estimation. In the extreme case, if we treat every point as a single gaussian center, KDE can also be thought as a special case of MoG model. Roughly speaking, as we increase the number of Gaussians in MoG model, the sensitivity increases, which implies the segmentation of object becomes more and more accurate.

However, there is big issue of MoG, that is the accuracy does not converge with the number of training examples. That is mainly because to best fit data into MoG model, an Expectation-Maximization (EM) algorithm is used. EM attempts to optimize some non-convex function by iterations. It is not always possible to find a good optimal solution. Bad initializations of EM would make the algorithm get trapped around some local optima. In fact, in our experiment, some of our runs stopped after 100 iterations without convergence. As a result, the MoG fitting is likely to be unreliable. On the other hand, for a mixture of m Gaussians, the complexity of EM is $\mathcal{O}(mn)$ for every iteration. In a extreme case, if the mixture number is equal to the number of points, the complexity is then $\mathcal{O}(n^2a)$, where a is the number of iterations. a can be large, e.g., in our experiment, it can be larger than 100, to converge, especially in a high dimensional data space.

Compared to the KDE model, which require $\mathcal{O}((n \log n + nk_t + nk_t^2) \log k_{t,max})$ in cross validation, the training of MoG is $\mathcal{O}(nma)$. Usually, the dominating term of KDE is $\mathcal{O}(nk_t^2 \log k_{t,max})$. To fig the distribution of data well, we might need a large number of Gaussians, namely m is large, and EM requires lots of iterations to converge (a is large). It is difficult to say which one between ma and $k_t^2 \log k_{t,max}$ is larger. Furthermore, It is also possible EM would get trapped at some local optima and totally failed. However, KDE do not have the local optima problem. As long as we have sufficient data, KDE would converge to the true distribution of data [120].

Table 21: *Likelihood with Different Probability Estimators:* $mogk$ indicates mixture of k gaussians model. #trn is the number of training images. To clearly illustrate the effects of different distribution estimator, the error rate is evaluated based on the likelihood term only without any smoothing or other priors.

#mog \ #trn	J^{ob}				J^{bg}			
	10	*50*	*100*	*500*	*10*	*50*	*100*	*500*
mog1	0.2796	0.2232	0.2150	0.2092	0.7869	0.6648	0.6419	0.6267
mog5	0.2619	0.6211	0.5578	0.5658	0.9329	0.9460	0.9215	0.9244
mog10	0.1583	0.6960	0.6995	0.6053	0.9243	0.9666	0.9603	0.9345
mog50	0.0000	0.4466	0.6859	0.7614	0.9112	0.9488	0.9697	0.9714
mog100	0.0000	0.3865	0.6394	0.7939	0.9112	0.9436	0.9661	0.9788
kde	0.7863	0.8585	0.8850	0.9202	0.9715	0.9832	0.9876	0.9919

Finally, Figure 32 shows how k_ρ in the balloon KDE would affect the estimation of the likelihood. Usually, the number of nearest neighbors k_ρ is empirically chosen according to the dataset size n. In our experiments, we found the final weight is not sensitive to k_ρ in a relatively large range. Thus our automatic approach to choose k_ρ is sufficient, and we do not need to tune k_ρ with lots of efforts. Figure 32 gives some examples of the weight maps with different k_ρ. There is no obvious difference among those results.

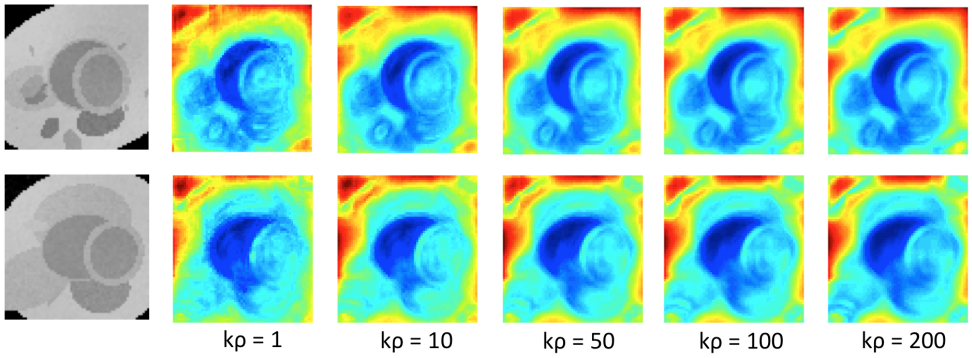


Figure 32: *Likelihood with different k_ρ:* The likelihood probability is estimated by the balloon KDE using 1000 training images, and fixed $k_t = 500$. It is not easy to observe some obvious difference of likelihood term.

4.4.4 Feature Selection

It is interesting to pay some attention in how to select Gabor features. Essentially, each dimension of Gabor features uses a Gabor filter with special window size, frequency and orientation. There could be redundancy between different dimensions. Some works were proposed to reduce the redundancy of Gabor features, such as carefully chosen parameters [88], PCA [133, 85], and machine learning techniques such as AdaBoost [118]. Machine learning techniques usually require expensive computational effort. As [88] points out, when increasing the redundancy, at least it would not do harm to the results. Due to the low dimensional structure of Gabor features, our kNN method converges fast, which make the fast evaluation of KDE feasible. Table 22 gives the accuracy using different set of Gabor features which constructed under different window sizes. It is shown that as we include more features, the accuracy is improved in terms of Jaccard index.

4.4.5 Methods comparison

Now we compared our approach with three frequently used segmentation methods: mean shift [26], graph cut [15, 54] and level set based approach [81]. We used EDISON [26] to apply mean shift, and used the matlab wrapper of graph cut algorithm provided by

Table 22: Accuracy given by likelihood term with different dimensional Gabor features: this table illustrates the segmentation accuracies given by the likelihood term only with respect to different nose level. We tested the likelihood with Gabor features with different scales (the largest window is the same as the input image), yielding features from dimensionality 32 to 176. Under each scale, we use 8 orientations, and the frequency of Gabor filter adaptively chosen according to the scales. It shows that as the dimensionality of features increases, the accuracy would improve as well.

#scales / noise	J^{ob}				J^{bg}			
	1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
	d=32	d=72	d=120	d=172	d=32	d=72	d=120	d=172
0%	0.4839	0.6193	0.6991	0.8771	0.9110	0.9521	0.9648	0.9875
1%	0.4503	0.5928	0.6872	0.8733	0.9031	0.9486	0.9634	0.9871
10%	0.3513	0.4540	0.6063	0.8456	0.8728	0.9277	0.9530	0.9841
20%	0.3087	0.3774	0.5447	0.8187	0.8573	0.9151	0.9441	0.9811

Bagon [8, 14, 15]. The distance regularized level set code is provided by [81]. The internal parameters in these software were carefully chosen to give good results.

Table 23 illustrates the accuracy of those methods. All three methods used image intensity directly. Mean shift and graph cut both need to cluster pixels into different groups, where we can use Gabor features though. The problem is if we do not have enough data, like mean shift and graph cut, which only cluster pixels within a single image, it is possible not to recover the true intrinsic manifold structure of data in the high dimensional feature space, hence the direct distance might not be correct to measure the similarity among points. The results of Gabor features are even worse than using intensities directly in our experiments. For level set, Gabor features are not straightforward to use. We may apply level set on every dimension, and then merge the results together. However it is unclear how to merge the results, and the merging weight on each dimension is related to the feature selection problem, which is still challenging. Furthermore, improper initialization may crash the whole level set algorithm.

Compared to our method, only mean shift is comparable in the case that there is no

noise exist in the data. If there are some noise, even only 1%, the accuracy degenerate quickly. In other cases, none of these three approaches obtain as good as our method. And our method suffer much less from noise.

Table 23: *Segmentation accuracy with mean-shift, graph-cut and level set methods*: this table gives the segmentation accuracy using several frequently used methods. Mean-shift results are obtained by software (EDISON) [26]. The smoothing bandwidths in feature space and spatial space are both set to be 5; the minimum segment area is set to be 100. The graph cut uses the software provided by [54]. The cluster number is set to be 4. The level set method used the software provided by Chunming Li [81]. All of these three approaches were applied directly to the input image rather than the Gabor features.

#scales / noise	J^{ob}				J^{bg}			
	0%	*1%*	*10%*	*20%*	*0%*	*1%*	*10%*	*20%*
mean shift [26]	0.9037	0.8446	0.5962	0.4804	0.9819	0.9748	0.9427	0.9302
graph cut [54]	0.3320	0.3248	0.2642	0.2245	0.7805	0.7901	0.7762	0.7593
level set [81]	0.5631	0.6405	0.6401	0.5745	0.6559	0.7728	0.9010	0.9029
our method	0.9291	0.9167	0.8883	0.8613	0.9930	0.9916	0.9885	0.9853

4.4.6 Segmentation of brain MR images

We also apply our method to brain MR images. The brain MR images we used can be categorized into three groups: easy (with little blurriness and noise), hard (with moderate blurriness and noise) and hardest (with large blurriness and noise). All the brain images are of size 256×256. Table 24 gives the segmentation accuracy using 1 to 1000 training images in three groups respectively. Generally speaking, the accuracy converges as the number of training images increases. However, compared with the synthesized images, it converges slower. That is mainly because the structures of brain tissues are way more complicated. Compared to white matter, grey matter is harder to segment. That is because the grey matter and the background have similar intensity and usually they are blurred together. Figure 34 gives some segmentation results. In fact, even in the hardest case, although it is even difficult to tell the difference between the white and grey matter by human eyes, our

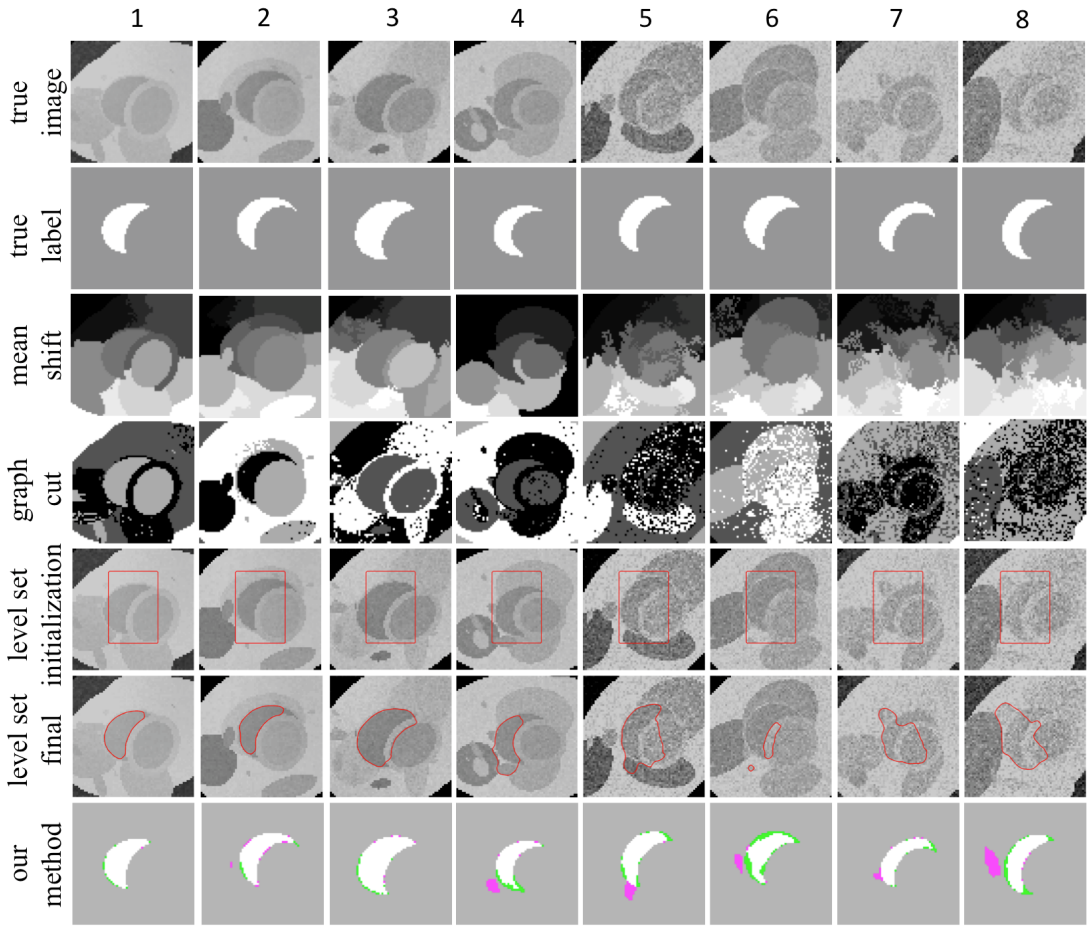


Figure 33: *Examples of segmentation by different approaches*: We compare our method with three frequently used segmentation methods. The **row (a)** shows the testing images we have used to test our algorithm. The **row (b)** gives the corresponding ground truth segmentations of the objects. The **row (c)** depicts the mean shift results. Generally speaking, mean shift cannot specify the exact number of clusters, and it only segmented images into different connected regions according to the smoothed images. The **row (d)** depicts the segmentations using the graph cuts algorithm. The **row (e)** gives the initializations of distance regularized level set algorithm, and the row **row (f)** shows the corresponding final zero level contours. Finally, the row **row (g)** gives the segmentation of our approach. Image 1 and 2 are clean images; image 3, 4 have 1% noise; image 5, 6 contains 10% noise; image 7 and 8 are of 20% noise. To indicate the difference between our segmentations of the true segmentations, we use '**white**' indicates the overlap region between the true label and our segmentation, '**green**' indicates the left region of the true label excluding the overlap, and '**magenta**' represents the remaining region of our segmentation besides the overlap.

approach gives some meaningful segmentations.

Table 24: *Segmentation accuracy of brain images*: this table gives the segmentation accuracy for brain images by KDE. We use Gabor features of four scales and eight orientations. we have three groups of images: with little blurriness and noise (easy), with moderate blurriness and noise (hard), and with large blurriness and noise (hardest).

		J^{ob}				J^{bg}			
#trn		*1*	*10*	*100*	*1000*	*1*	*10*	*100*	*1000*
grey matter	easy	0.6298	0.6596	0.6952	0.7173	0.9434	0.9500	0.9579	0.9621
	hard	0.5347	0.5444	0.5803	0.6046	0.9283	0.9313	0.9387	0.9438
	hardest	0.3493	0.3571	0.3640	0.4017	0.8972	0.8963	0.9003	0.9061
white matter	easy	0.7494	0.7781	0.8183	0.8363	0.9646	0.9682	0.9229	0.9760
	hard	0.6862	0.7046	0.7438	0.7600	0.9520	0.9537	0.9593	0.9628
	hardest	0.5442	0.5381	0.5626	0.5822	0.9170	0.9115	0.9235	0.9299

4.4.6.1 Comparison with SVM

In this subsection, we compare the our approach with one frequently used classifier: support vector machine (SVM). In our experiment, we use LIBLINEAR [48], which implements a linear SVM classifier. We do not use nonlinear kernels because training a non linear SVM requires solving a quadratic programming, each iteration takes$\mathcal{O}(n^2)$. If n is relatively large, using cross validation to choose proper parameters is very time consuming and impractical in real applications. Table 25 gives results using SVM to classify grey matter and white matter. Unfortunately, the grey matter seems to be non linear separable. SVM fails to correctly label grey matter. However, if the object is linear separable, SVM outperform over KDE approach. SVM needs less training data to obtain a similar accuracy of white matter. To obtain a similar accuracy, KDE need more training data. Compared with KDE based approach, the main shortage of SVM is the expensive time complexity. If we use linear SVM, for some non linearly separable case, it would fail.

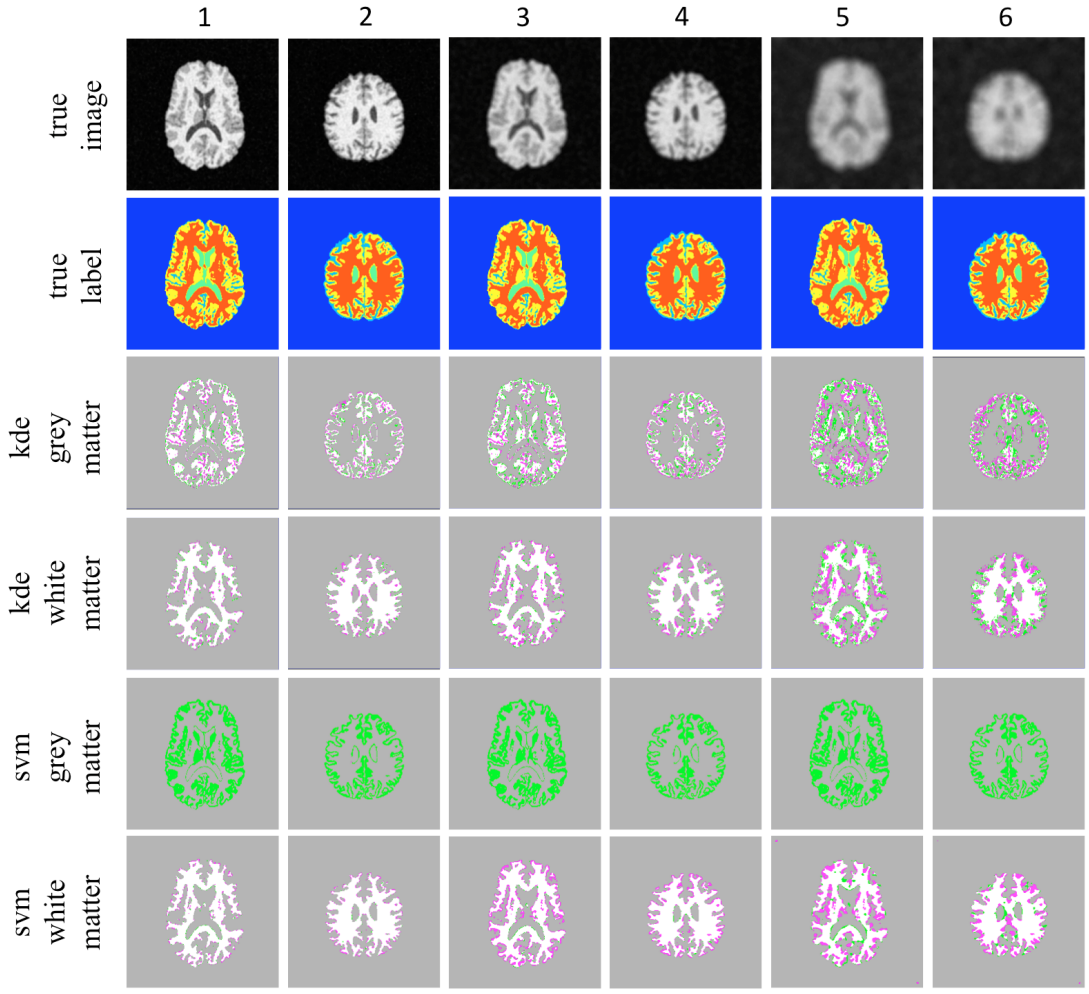


Figure 34: *Examples of brain segmentation*: We try to segment the grey matter and white matter for a given brain MR image. All figures are of size 256×256. In this experiment, all segmentations use 1000 training images and Gabor features of four scales and eight orientations. The first two columns are normal brain images, the middle two columns are brain images with some blurriness and noise. The last two columns are images with more blurriness and noise. In the second row, the yellow color indicates the grey matter and the orange color represents the white matter region.

Table 25: *Segmentation accuracy of brain images*: this table gives the segmentation accuracy for brain images obtained by SVM. We use Gabor features of four scales and eight orientations. we have three groups of images: with little blurriness and noise (easy), with moderate blurriness and noise (hard), and with large blurriness and noise (hardest). To indicate the difference between our segmentations of the true segmentations, we use '**white**' indicates the overlap region between the true label and our segmentation, '**green**' indicates the left region of the true label excluding the overlap, and '**magenta**' represents the remaining region of our segmentation besides the overlap.

		J^{ob}			J^{bg}		
#trn		*1*	*10*	*100*	*1*	*10*	*100*
grey matter	easy	0.0699	0.0045	0.0000	0.8846	0.8968	0.8969
	hard	0.0790	0.0042	0.0000	0.8793	0.8968	0.8969
	hardest	0.0622	0.0000	0.0000	0.8804	0.8969	0.8969
white matter	easy	0.8486	0.8494	0.8527	0.9801	0.9796	0.9804
	hard	0.7743	0.7772	0.7770	0.9666	0.9662	0.9661
	hardest	0.6346	0.6471	0.6523	0.9391	0.9436	0.9432

4.5 Conclusions

We proposed a Bayesian variational image segmentation scheme that uses a high-dimensional KDE-based approximation of the likelihood function (we showed examples with over 300 dimensions with 4 million points). Our approach is possible only due to recent advances in fast exact and approximate algorithms for KDE in high dimensions. Also, the proposed method makes sense only in the presence of a lower-dimensional manifold, which has been experimentally observed by many groups.

One question is why not using this KDE approach to construct the prior. One complication is that introduces significant non-convexity. However, this is an important research direction that we will explore in the future. Another important aspect is the issue of scaling and appropriate alignment of the training samples, which we do not discuss in detail here.

Overall the method in the barebones formulation we described here performs quite well,

is fast, (requires O(1) work per pixel), results in a unique MAP solution, and has no arbitrarily selected constants. The main parameters are the KDE bandwidth and the smoothness regularization which are both selected using the training set and cross validation.

Although our model is built around a wrong assumption, the pixel-wise conditional independence assumption of the features, the segmentation quality does improve with increasing sample size. It will be interesting to devise a theoretical understanding of under what conditions our model approximates the true likelihood function well.

CHAPTER V

STATISTICAL SPACIAL PRIOR

In chapter 4, we discussed the likelihood function and experimentally showed the affects of various parameters in terms of segmentation errors. In this chapter, we describe the prior function, combining the likelihood to finish the MAP estimation framework. The prior can be also used as a single regularization term which can pose shape constraint on various inverse problems such as the least square denoising, registration, etc.. Unlike some existing works which treat each image as a whole data point and requires image alignment as a preprocessing, our new statistical shape prior do not need any registration to align images into the same coordinate system, then we do not couple two hard problems, segmentation and registration together. In §5.2, we discuss the problem formulation and §5.3 describes the numerical implementation of the fast prior evaluation and optimization. The convergence and complexity of the optimization is discussed in §5.4. §5.5 demonstrates the performance of our new prior in two aspects, one is used as a regularization term in denoising, the other is the prior in image segmentation.

5.1 Introduction

Segment medical images is a very challenge and ill-posed task. Usually people only consider the information of statistics on the intensity of two regions, the background and the object, which do not take any consideration of object shapes. However, shape priors can be quite successful in compensating for misleading information due to noise, artifacts, occlusion in the input image. Recently, incorporation of prior knowledge on the shape of the segmenting objects or contours has generated lots of research interests. A common idea is to learn the shape of an object statistically from a set of training shapes, and then constrain

the segmented contour or object to a sub manifold of the known shapes during the optimization (evolving) process. For the problem of segmenting a known object priors were demonstrated to improve the accuracy significantly [79, 35].

5.1.1 Related work

The goal of building shape models is to infer and refine the object shape from the existing approaches in an optimal sense. Some of the pioneer works are the 'Snake' and 'ASM', which models the shape prior as a general regularization term in the optimization and conform the shape should deform like a thin plate or the training shapes [73, 29].

Many adaptations of these algorithms have been proposed over yeas. Roughly speaking, shapes are assumed sampled from a multivariate Gaussian distribution, which essentially implies that all possible shape deformations correspond to linear combinations of a set of eigen-shapes obtained by principal component analysis [29, 79, 31]. However, their work is limited to a linearized shape space with small deformation modes around a mean shape. Usually the mean shape is formulated as an optimal solution to minimize a certain cost function. Different cost functions lead to different optimization framework. However due to noise, clutter and intra class variations of the shape, it is difficult to find a tractable global minimum [27, 67, 36, 20]. More efforts have been put on handling multimodal distribution of shapes, where shapes cannot be reconstructed from the mean shape and its variations. A classical extension is to model shape distribution as a mixture of Gaussians [28]. Shapes also be defined as a finite dimensional manifold by diffusion maps, and each shape can be projected to one point on the manifold [47]. In order to preserve the local details which are not statistically significant, people also use sparse linear combination of training shapes to model complex shape variations [46, 136]. Other works include patient specific shape statistics [119] or subject specific model [138] to constrain the deformable contours. This works for the case that each image has a particular shape even if the mean shape cannot capture the characteristic of target objects.

There are also many works using Mumford-Shah functional or Chan-Vese functional which are to find a piecewise smooth function which approximates the image [94, 17]. Lots of attempts have already been tried to combine shape priors as a second energy term to simultaneously optimize with those functionals [112, 18, 32]. Either the mean shape way [79] or a more statistical way using kernel density estimation to the domain of level set based shape representations [112, 32] were used.

To define a shape, one question is how to represent the training images. A frequently used one is the edge points [20, 27, 36, 112]. However, edge point is not that informative for local variations and the cost functions using edge points have lots of local minima. More complex features are also proposed to represent shapes such as shape context [9], PAS [49] or edge let [131]. In the variational framework, people also use the signed distance function from level set as the shape representation and use different distance metric to define label functions [79, 18, 32].

5.1.2 Our method

We consider a new statistical prior which does not need registration as preprocessing, and our new prior, unlike most existing works which evolve from a certain mean shape to a specific object, can be capable to deal with more complicated variations of shapes. Rather than using edge points which are sensitive to the noise and artifacts, and the signed distance functions, which is computationally expensive to obtain and also venerable to image clutters, we use a simple wavelet features which is proven to be information about the spacial correlations and insensitive to noise in chapter 4.

The same as the likelihood in chapter 4, the prior is also defined using the non-parametric nearest neighbor KDE. To simplify the calculation, we also assume that the Gabor features of each pixel conditioned on the label of the pixel are independently and identically distributed (i.i.d). To find the MAP estimation, we relax the integrality condition for ϕ and we only require that $\phi \in [0,1]^N$, and we solve an inequality-constrained quadratic program to

compute ϕ. Moreover, a fast way to evaluate the gradient and Hessian is also established for the minimization of our functional.

5.1.3 Contributions

We present a new spacial prior and its fast evaluation scheme for optimization. This new prior can be used for image segmentation as well as other general inverse problem such as image denoising and deformable registration. In particular,

- *New shape prior without requirement of alignment.* No matter the mean shape or the signed distance functions, all of them require the images to be aligned into the same coordinate system before calculating the shape priors. However, image registration is also a challenging problem and sometimes itself ask for segmentation informations.

  Our prior is easy to embed into any inference framework and inverse problems such as registration and denoting as well.

- *Fast evaluation of Gradient and Hessian matrix of cost functions.* We use the TRON [84] algorithm to solve the optimization. In order to accelerate the calculation, we proposed the fast evaluation of Gradient and Hessian based on FFT, and all the computations are parallelized.

- *Robust to noise and artifacts.* Experimental evidences show the new prior works well for the incomplete least square denoising problem and outperform most of the existing regularization term. It also improve the likelihood functions in image segmentation results.

5.1.4 Limitations

Like the likelihood in chapter 4, we also made the assumption that the pixel-wise values of the features are i.i.d, which is in correct. Besides, the requirement for a large set of training images potentially limits the applicability of the proposed prior method.

5.2 Problem Formulation

As we mentioned in the introduction, $\mathbf{I}_i$ and $\boldsymbol{\phi}_i, i = 1, \cdots, N$ indicate the discrete grayscale image and label value at the pixel i, where i is the index on a 2D lattice and N is the product of width and height of this lattice. Assume

$$\mathbf{I} = f(\boldsymbol{\phi}) + \varepsilon \tag{46}$$

where $\mathbf{I}$ is the observed target image, and $f(\boldsymbol{\phi})$ is the function value where $f(\cdot)$ describes a forward physical process that how is the observed image $\mathbf{I}$ generated from a source $\boldsymbol{\phi}$. We also have assumed that the observed value $\mathbf{I}$ differs from the function value $f(\boldsymbol{\phi})$ by certain additive noise.

Given the observed image $\mathbf{I}$, usually people want to infer the label $\boldsymbol{\phi}$ from the posterior distribution $p(\boldsymbol{\phi}|\mathbf{I}) \sim p(\mathbf{I}|\boldsymbol{\phi})p(\boldsymbol{\phi})$. The Maximum a posteriori (MAP) estimate is found by maximizing the log-posteriori:

$$\boldsymbol{\phi}^* = \text{argmax}_{\boldsymbol{\phi}} \log p(\mathbf{I}|\boldsymbol{\phi}) + \log p(\boldsymbol{\phi}) = \text{argmin}_{\boldsymbol{\phi}} E_l(\boldsymbol{\phi}) + E_p(\boldsymbol{\phi}), \tag{47}$$

where $E_l(\phi) = -\log p(\mathbf{I}|\boldsymbol{\phi})$ is the energy of likelihood term and $E_p(\boldsymbol{\phi}) = -\log p(\boldsymbol{\phi})$ is the energy of prior term. Typically $p(\boldsymbol{\phi})$ is given as a Gibbs distribution $\sim e^{-V(\boldsymbol{\phi})}$. Then we can set $E_p(\boldsymbol{\phi}) = V(\boldsymbol{\phi})$ since the proportionality constant is not needed in the computation of $\boldsymbol{\phi}^*$.

5.2.1 Modeling forward process $f(\boldsymbol{\phi})$

We can directly model the physical process $f(\boldsymbol{\phi})$ according to the corresponding problems. For example, in a standard linear model

$$f(\boldsymbol{\phi}) = \mathbf{A}\boldsymbol{\phi}, \;\; \mathbf{I} = \mathbf{A}\boldsymbol{\phi} + \varepsilon \tag{48}$$

where $\mathbf{A}$ is known but can be highly non-singular.

Assume the additive noise follows an independent, identically distributed (i.i.d.) Gaussian distribution with zero mean and variance σ_n^2, that is $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$. This noise assumption together with the model gives the likelihood

$$p(\mathbf{I}|\boldsymbol{\phi}, \mathbf{A}) = \frac{1}{\sqrt{2\pi}\sigma_n} \exp\left(-\frac{\|\mathbf{I} - f(\boldsymbol{\phi})\|^2}{2\sigma_n^2}\right) \tag{49}$$

Thus the minimization problem in Eqn. (47) becomes

$$\boldsymbol{\phi}^* = \text{argmin}_{\boldsymbol{\phi}} \frac{1}{2\sigma_n^2} \left(\mathbf{I} - f(\boldsymbol{\phi})\right)^T \left(\mathbf{I} - f(\boldsymbol{\phi})\right) + E_p(\boldsymbol{\phi}) \tag{50}$$

The object function in Eqn. (50) is a typical least square problem and $E_p(\boldsymbol{\phi})$ can be treated as a regularization term. There are other more complicated nonlinear models such as various partial differential equations to model the forward process $f(\boldsymbol{\phi})$ as well [65].

5.2.2 Modeling likelihood function $p(\mathbf{I}|\boldsymbol{\phi})$

In many practical applications, however, the function $f(\boldsymbol{\phi})$ in Eqn. (46) is unknown. Instead of making assumption on the forward generation function $f(\boldsymbol{\phi})$, people can directly model the likelihood functions. A typical assumption for the likelihood function is that the conditional intensity is i.i.d., that is

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{i=1}^{N} p(\mathbf{I}_i|\boldsymbol{\phi}_i). \tag{51}$$

Typically, a parametric density is assumed for $p(\mathbf{I}_i|\boldsymbol{\phi}_i)$ and the parameters are estimated using the training set. Being a histogram-based model, the KDE has a strong assumption that pixels are independent, where spatial information is not taken into consideration. For images, at least nearby pixels are highly correlated with each other. Rather than consider pixels, we first define the feature map $\mathbf{G} : \mathrm{R}^N \rightarrow \mathrm{R}^{d \times N}$ that maps scalar intensities to a $d-$dimensional vectors so that $\mathbf{g}_i = \mathbf{G}(\mathbf{I})_i \in \mathcal{R}^d$ is the feature vector of pixel i. To define the likelihood, we assume that *the features* are conditionally independent so that

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{i=1}^{N} p_i(\mathbf{I}) = \prod_{i=1}^{N} p(\mathbf{g}_i|\boldsymbol{\phi}_i) = \prod_{i=1}^{N} p(\mathbf{G}(\mathbf{I})_i|\boldsymbol{\phi}_i). \tag{52}$$

where $p_i(\mathbf{I}) = p(\mathbf{G}(\mathbf{I})_i|\boldsymbol{\phi}_i)$

For a binary segmentation to "*object*" ($\boldsymbol{\phi}_i = 1$) and "*background*" ($\boldsymbol{\phi}_i = 0$), Eqn. (52) simplifies to

$$p(\mathbf{I}|\boldsymbol{\phi}) = \prod_{i:\boldsymbol{\phi}_i=1} p_i^{obj}(\mathbf{I}) \prod_{i:\boldsymbol{\phi}_i=0} p_i^{bg}(\mathbf{I}) = \prod_{i=1}^{N} p_i^{obj}(\mathbf{I})^{\boldsymbol{\phi}_i} p_i^{bg}(\mathbf{I})^{1-\boldsymbol{\phi}_i} \tag{53}$$

where $p_i^{obj}(\mathbf{I}) \equiv p\left(\mathbf{G}(\mathbf{I})_i|\boldsymbol{\phi}_i = 1\right)$ and $p_i^{bg}(\mathbf{I}) \equiv p\left(\mathbf{G}(\mathbf{I})_i|\boldsymbol{\phi}_i = 0\right)$. With this notation, the log-likelihood can be written as

$$E_l = \sum_{i=1}^{N} \boldsymbol{\phi}_i \log p_i^{obj}(\mathbf{I}) + (1 - \boldsymbol{\phi}_i) \log p_i^{bg}(\mathbf{I}) \tag{54}$$

In the infinite dimensional setting (which is convenient for defining the prior), the above log-likelihood for an image function $I(x)$ and a label function $\phi(x)$, becomes

$$\begin{aligned} E_l &= \int_x \phi(x) \log p^{obj}(x) + (1 - \phi(x)) \log p^{bg}(x) \\ &= \int_x \phi(x) \log \frac{p^{obj}(x)}{p^{bg}(x)} + \int_x \log p^{bg}(x) \end{aligned} \tag{55}$$

where the term $\int_x \log p^{bg}(x)$ only depends on the observed image itself. For any given image $\mathbf{I}$, it is a fixed constant.

For the discretized case, $E_l = \mathbf{a} \cdot \boldsymbol{\phi} + const$, where

$$\mathbf{a}_i = \log p_i^{obj}(\mathbf{I}) - \log p_i^{bg}(\mathbf{I}). \tag{56}$$

Given this definition of likelihood, we need to decide on the approximation of $p^{obj}(i)$ and $p^{bg}(i)$. We use a kernel density estimation approach to approximate those probabilities. The details will be discussed in §4.2.

5.2.3 Modeling prior $p(\boldsymbol{\phi})$

In Eqn. (47), we need a shape prior $p(\boldsymbol{\phi})$ to complete the MAP estimation. A frequently used prior is the Gaussian assumption [23, 21]. Each $\boldsymbol{\phi}$ is one high dimensional point and all $\boldsymbol{\phi}^j$'s follow a Gaussian or Mixture of Gaussians (MoG) distributions, that is

$$p(\boldsymbol{\phi}) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}\left(\boldsymbol{\phi} - \bar{\boldsymbol{\phi}}\right)^T \boldsymbol{\Sigma}^{-1} \left(\boldsymbol{\phi} - \bar{\boldsymbol{\phi}}\right)\right) \tag{57}$$

Another simple prior is the smoothness assumption. In the infinite-dimensional setting, we define the prior by setting

$$V(\phi) = \frac{\beta}{2} \int_x \nabla\phi \cdot \nabla\phi \tag{58}$$

which upon discretization becomes $V(\boldsymbol{\phi}) = \frac{1}{2}\boldsymbol{\phi} \cdot \mathbf{L}_\beta \boldsymbol{\phi}$, where $\mathbf{L}_\beta$ is the discrete Laplacian weighted by $1/\beta$, a regularization parameter that is related on the level of noise. In this work, a simple five-point-stencil Laplacian with Neumann conditions was used to define the Laplacian operator.

A more practical way is similar to §5.2.2, under the assumption that features at each position are independent, we have

$$p(\boldsymbol{\phi}) = \prod_{i=1}^{N} p_i(\boldsymbol{\phi}) = \prod_{i=1}^{N} p_i^{obj}(\boldsymbol{\phi})^{\boldsymbol{\phi}_i} \prod_{i=1}^{N} p_i^{bg}(\boldsymbol{\phi})^{(1-\boldsymbol{\phi}_i)} \tag{59}$$

where $p_i^{obj}(\boldsymbol{\phi}) \equiv p\left(\mathbf{G}(\boldsymbol{\phi})_i | \boldsymbol{\phi}_i = 1\right)$ and $p_i^{bg}(\boldsymbol{\phi}) \equiv p\left(\mathbf{G}(\boldsymbol{\phi})_i | \boldsymbol{\phi}_i = 0\right)$. The energy of prior then has the form

$$E_p(\boldsymbol{\phi}) = -\log p(\boldsymbol{\phi}) = -\sum_{i=1}^{N} \boldsymbol{\phi}_i \log p_i^{obj}(\boldsymbol{\phi}) - \sum_{i=1}^{N} (1-\boldsymbol{\phi}_i) \log p_i^{bg}(\boldsymbol{\phi}) \tag{60}$$

5.2.4 Solving ϕ by Optimization

Finally, we can solve the optimal $\boldsymbol{\phi}$ by minimizing Eqn. (47) by combining the prior and likelihood term. In our binary case, $\boldsymbol{\phi}_i$ is either 0 or 1. Solving an integer programming is always difficult especially the object function is non convex. We relax $\boldsymbol{\phi}_i \in [0, 1]$ to make the optimization a continuous problem. In a perspective of modeling function $f(\boldsymbol{\phi})$ directly (§5.2.1), we have

$$\begin{aligned} \boldsymbol{\phi}^* = \quad & \text{argmin}_{\boldsymbol{\phi}} \quad \|\mathbf{I} - f(\boldsymbol{\phi})\|^2 - \sum_{i=1}^{N} \boldsymbol{\phi}_i \log p_i^{obj}(\boldsymbol{\phi}) - \sum_{i=1}^{N} (1-\boldsymbol{\phi}_i) \log p_i^{bg}(\boldsymbol{\phi}) \\ & s.t. \qquad 0 \le \boldsymbol{\phi}_i \le 1 \end{aligned} \tag{61}$$

If we model the likelihood $p(\mathbf{I}|\boldsymbol{\phi})$, then

$$\begin{aligned} \boldsymbol{\phi}^* = \quad & \text{argmin}_{\boldsymbol{\phi}} \quad \mathbf{a}^T\boldsymbol{\phi} - \sum_{i=1}^{N} \boldsymbol{\phi}_i \log p_i^{obj}(\boldsymbol{\phi}) - \sum_{i=1}^{N} (1-\boldsymbol{\phi}_i) \log p_i^{bg}(\boldsymbol{\phi}) \\ & s.t. \qquad 0 \le \boldsymbol{\phi}_i \le 1 \end{aligned} \tag{62}$$

where $\mathbf{a}$ is defined in Eqn. (56).

Both Eqn. (61) and Eqn. (62) are box constrained non convex optimization problems. To solve them, we can apply TRON algorithm [84]. Each iteration of the TRON algorithm requires function, gradient and Hessian evaluation. We would discuss all the numerical details in §5.3.

5.3 Numerical Algorithms

In this section, we first we describe a fast calculation of KDE, and then introduce how to evaluate the gradient and Hessian for every iteration in TRON. Finally, we will discuss the parallelization of our algorithm.

5.3.1 Fast Calculation for KDE

In §4.2, we model the pixel-wise likelihood probability by the KDE which is approximated by KNN truncation. Here we use the same way to approximate the prior by only adding the first k_t nearest points as well.

$$p_i^c(\mathbf{I}) \approx \frac{1}{k_t} \sum_{j \in \{1,\cdots,k_t, \boldsymbol{\phi}_j = c\}} (2\pi)^{-d/2} (h_i)^{-d} \exp\left(\frac{\|\mathbf{G}_i \mathbf{I} - \mathbf{g}^j\|^2}{2h_i^2}\right) \tag{63}$$

$$p_i^c(\boldsymbol{\phi}) \approx \frac{1}{k_t} \sum_{j \in \{1,\cdots,k_t, \boldsymbol{\phi}_j = c\}} (2\pi)^{-d/2} (h_i)^{-d} \exp\left(\frac{\|\mathbf{G}_i \boldsymbol{\phi} - \mathbf{g}^j\|^2}{2h_i^2}\right) \tag{64}$$

where h_i is the bandwidth for each pixel i and d is the dimensionality of feature $\mathbf{g}^j$.

Two issues need to be solved to calculate Eqn. (63) and Eqn. (64): the first is how to find the k-nearest neighbors. The naive k-nearest neighbors search costs the same $\mathcal{O}(Nn_s)$ as the direct kernel summation. To accelerate computations, We use a parallel random projection tree algorithm to reduce the complexity to $\mathcal{O}(N \log n_s)$[68]. The basic idea is after hierarchically partitioning the reference space, we only search for part of the data in terms of leaf node on a binary tree. To increase the accuracy, we build several trees by rotating reference points at each time, and then aggregate the results of every single search. Details of this algorithm and parallel scalability can be found in chapter 2.

The second question is how to determine the bandwidth parameter k_h and truncation parameter k_t. A frequently used approach is cross validation (CV). Lots of objection functions can be applied for CV, including maximum likelihood loss function [44], unbiased least square [13, 130], biased least square [116], completed cross validation [104], etc.. In our problem, our focus actually is not the real probability density, but the accuracy of the recovered $\boldsymbol{\phi}$. Thus we use a different object function to search for k_h and k_t. For every pixel, we can have two probabilities $p_i^{obj}(\cdot)$ and $p_i^{bg}(\cdot)$. Ideally if $p_i^{obj}(\cdot) > p_i^{bg}(\cdot)$, the pixel i is treated as object, i.e., $\boldsymbol{\phi}_i$ should equal to 1. similarly, $p_i^{obj}(\cdot) < p_i^{bg}(\cdot)$ implies $\boldsymbol{\phi}_i = 0$. Define $\omega_i = (p_i^{obj}(\cdot) - p_i^{bg}(\cdot)) \times (\boldsymbol{\phi}_i - 0.5)$, we know that if $\omega_i > 0$, the probability given by the selected bandwidth conforms with $\boldsymbol{\phi}_i$, otherwise, the selected bandwidth gives improper $p_i^c(\cdot)$. The choice of bandwidth should penalize those pixels with $\omega_i < 0$ to make the probability in accordance with $\boldsymbol{\phi}$ as well as possible. Thus we use a logistic loss function

$$\begin{aligned} J(k_h, k_t) &= \sum_{i=1}^{N} \log(1 + \exp(-\omega_i)) \\ &= \sum_{i=1}^{N} \log\left(1 + \exp\left((p_i^{obj}(\cdot) - p_i^{bg}(\cdot)) \times (0.5 - \boldsymbol{\phi}_i)\right)\right) \end{aligned} \tag{65}$$

Algorithm 19 KLCV

initial k_t, ξ to make the CV start
find k_t near neighbors for every $\mathbf{g}^j$.
find $k_h^* = \min_{k_h} J(k_h, k_t), k_h = 1, 2, \cdots, k_t$
$J_{old} = J(k_h^*, k_t)$
while $\xi > \eta \ \ and \ \ k_t < k_{t,max}$ **do**
    $k_t = 2k_t$
    find k_t near neighbors for every $\mathbf{g}^j$
    find $k_h^* = \min_k J(k_h, k_t)$
    $\xi = |J(k_h^*, k_t) - J_{old}|/J_{old}$
    $J_{old} = J(k_h^*, k_t)$
end while
return k_h^*, k_t

5.3.2 Fast Operations for General Loss Function

We assume the loss functions in Eqn. (61) and Eqn. (62) are general twice-differentiable functions. The nonlinear part is the energy part (Eqn. (60)). Define

$$K_{ij} \equiv \exp\left(-\frac{(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)^T(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)}{2h_i^2}\right) \tag{66}$$

$$\mathbf{Z}_i^c := \frac{\sum_{j\in c}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)K_{ij}}{\sum_{j\in c}K_{ij}} \tag{67}$$

From the truncation KDE in Eqn. (64), we have

$$\begin{aligned}
\partial_{\boldsymbol{\phi}}\log p_i^c(\boldsymbol{\phi}) &= \frac{\frac{1}{k_t}\mathbf{G}_i^T\sum\limits_{j\in c}(2\pi)^{-d/2}h_i^{-d}K_{ij}\left(-\frac{1}{h_i^2}\right)(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)}{\frac{1}{k_t}\sum\limits_{j\in c}(2\pi)^{-d/2}h_i^{-d}K_{ij}} \\
&= -\frac{1}{h_i^2}\frac{\mathbf{G}_i^T\sum\limits_{j\in c}K_{ij}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)}{\sum\limits_{j\in c}K_{ij}} \\
&= -\frac{1}{h_i^2}\mathbf{G}_i^T\mathbf{Z}_i^c
\end{aligned} \tag{68}$$

$$\begin{aligned}
\partial_{\boldsymbol{\phi}\boldsymbol{\phi}}\log p_i^c(\boldsymbol{\phi}) &= \frac{\partial_{\boldsymbol{\phi}\boldsymbol{\phi}}p_i^c(\boldsymbol{\phi})}{p_i^c(\boldsymbol{\phi})} - \left[\frac{\partial_{\boldsymbol{\phi}}p_i^c(\boldsymbol{\phi})}{p_i^c(\boldsymbol{\phi})}\right]\left[\frac{\partial_{\boldsymbol{\phi}}p_i^c(\boldsymbol{\phi})}{p_i^c(\boldsymbol{\phi})}\right]^T \\
&= \mathbf{G}_i^T\frac{\sum\limits_{j\in c}K_{ij}\left[\frac{1}{h_i^4}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)^T-\frac{1}{h_i^2}\mathbf{E}\right]}{\sum\limits_{j\in c}K_{ij}}\mathbf{G}_i \\
&\quad -\frac{1}{h_i^4}\mathbf{G}_i^T\left[\frac{\sum\limits_{j\in c}K_{ij}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)}{\sum\limits_{j\in c}K_{ij}}\right]\left[\frac{\sum\limits_{j\in c}K_{ij}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)}{\sum\limits_{j\in c}K_{ij}}\right]^T\mathbf{G}_i \\
&= \frac{1}{h_i^4}\mathbf{G}_i^T\left[\frac{\sum\limits_{j\in c}K_{ij}(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)(\mathbf{G}_i\boldsymbol{\phi}-\mathbf{g}^j)^T}{\sum\limits_{j\in c}K_{ij}}-\mathbf{Z}_i^c\mathbf{Z}_i^{cT}-h_i^2\mathbf{E}\right]\mathbf{G}_i
\end{aligned} \tag{69}$$

Plugging Eqn. (68) and Eqn. (69), the gradient and Hessian matrix for $E_p(\boldsymbol{\phi})$ are

$$\begin{aligned}
\partial_{\boldsymbol{\phi}}E_p(\boldsymbol{\phi}) &= \sum_{i=1}^N\left\{\left[\log p_i^{bg}(\boldsymbol{\phi})-\log p_i^{obj}(\boldsymbol{\phi})\right]\mathbf{e}_i-\partial_{\boldsymbol{\phi}}\left[\log p_i^{obj}(\boldsymbol{\phi})\boldsymbol{\phi}_i+\log p_i^{bg}(\boldsymbol{\phi})(1-\boldsymbol{\phi}_i)\right]\right\} \\
&= \sum_{i=1}^N\mathbf{G}_i^T\left(\frac{\boldsymbol{\phi}_i}{h_i^2}\mathbf{Z}_i^{obj}+\frac{1-\boldsymbol{\phi}_i}{h_i^2}\mathbf{Z}_i^{bg}\right)+\sum_{i=1}^N\left[\log p_i^{bg}(\boldsymbol{\phi})-\log p_i^{obj}(\boldsymbol{\phi})\right]\mathbf{e}_i
\end{aligned} \tag{70}$$

$$
\begin{aligned}
\partial_{\boldsymbol{\phi}\boldsymbol{\phi}} E_p(\boldsymbol{\phi}) &= \sum_{i=1}^{N} \left\{ \left[\partial_{\boldsymbol{\phi}} \log p_i^{bg}(\boldsymbol{\phi}) - \partial_{\boldsymbol{\phi}} \log p_i^{obj}(\boldsymbol{\phi}) \right] \mathbf{e}_i^T + \mathbf{e}_i \left[\partial_{\boldsymbol{\phi}} \log p_i^{bg}(\boldsymbol{\phi}) - \partial_{\boldsymbol{\phi}} \log p_i^{obj}(\boldsymbol{\phi}) \right]^T \right\} \\
&\quad - \sum_{i=1}^{N} \left[\boldsymbol{\phi}_i \cdot \partial_{\boldsymbol{\phi}\boldsymbol{\phi}} \log p_i^{obj}(\boldsymbol{\phi}) + (1 - \boldsymbol{\phi}_i) \cdot \partial_{\boldsymbol{\phi}\boldsymbol{\phi}} \log p_i^{bg}(\boldsymbol{\phi}) \right] \\
&= \sum_{i=1}^{N} \mathbf{G}_i^T \left\{ \frac{\boldsymbol{\phi}_i}{h_i^4} \left(\mathbf{Z}_i^{obj} \right) \left(\mathbf{Z}_i^{obj} \right)^T + \frac{1 - \boldsymbol{\phi}_i}{h_i^4} \left(\mathbf{Z}_i^{bg} \right) \left(\mathbf{Z}_i^{bg} \right)^T + \frac{1}{h_i^2} \mathbf{E} \right\} \mathbf{G}_i \\
&\quad - \sum_{i=1}^{N} \mathbf{G}_i^T \left\{ \frac{\boldsymbol{\phi}_i}{h_i^4} \frac{\sum\limits_{j \in obj} K_{ij} (\mathbf{G}_i \boldsymbol{\phi} - \mathbf{g}^j)(\mathbf{G}_i \boldsymbol{\phi} - \mathbf{g}^j)^T}{\sum\limits_{j \in obj} K_{ij}} \right. \\
&\qquad \left. + \frac{1 - \boldsymbol{\phi}_i}{h_i^4} \frac{\sum\limits_{j \in bg} K_{ij} (\mathbf{G}_i \boldsymbol{\phi} - \mathbf{g}^j)(\mathbf{G}_i \boldsymbol{\phi} - \mathbf{g}^j)^T}{\sum\limits_{j \in bg} K_{ij}} \right\} \mathbf{G}_i \\
&\quad + \sum_{i=1}^{N} \mathbf{G}_i^T \left(\frac{1}{h_i^2} \right) \left(\mathbf{Z}_i^{obj} - \mathbf{Z}_i^{bg} \right) \mathbf{e}_i^T + \sum_{i=1}^{N} \mathbf{e}_i \left(\frac{1}{h_i^2} \right) \left(\mathbf{Z}_i^{obj} - \mathbf{Z}_i^{bg} \right)^T \mathbf{G}_i
\end{aligned} \tag{71}
$$

where $\mathbf{E}$ is the identity matrix and $\mathbf{e}_i$ is the unit vector of all zeros except the i-th element which is 1.

Assuming all the logarithms of probability $\log p_i^{obj}$, $\log p_i^{bg}$ as well as all $\mathbf{Z}_i$ have been pre calculated, a direct computation of $\partial_{\boldsymbol{\phi}} E_p(\boldsymbol{\phi})$ then requires $\mathcal{O}(N^2 d)$, and calculation of $\partial_{\boldsymbol{\phi}\boldsymbol{\phi}} E_p(\boldsymbol{\phi})$ by matrix multiplication cost $\mathcal{O}(N^4 d^3 + N^3 d^5)$. In next section, we would give the faster procedures to perform both operations.

5.3.3 Fast Calculation of the Gradient and the Hessian

The most time consuming part in evaluating the gradient and hessian in Eqn. (70) and Eqn. (71) is the summation of the product of $\mathbf{G}_i$ with another vector, i.e., $\sum_{i=1}^{N} \mathbf{G}_i^T \mathbf{v}_i$. The direct computation needs N matrix multiplications of $\mathbf{G}_i^T \in \mathbb{R}^{N \times d}$ and $\mathbf{v}_i \in \mathbb{R}^{d \times 1}$, which totally cost $\mathcal{O}(N^2 d)$. We propose a fast calculation based on Fast Fourier Transform (FFT) to reduce this calculation to $\mathcal{O}(N \log N d)$.

Expand the summation $\sum_{i=1}^{N} \mathbf{G}_i^T \mathbf{v}_i$ in a matrix form, we have

$$
\begin{aligned}
\sum_{i=1}^{N} \mathbf{G}_i^T v_i &= \sum_{i=1}^{N} \begin{pmatrix} \mathbf{G}_{i,(1,1)} & \mathbf{G}_{i,(2,1)} & \cdots & \mathbf{G}_{i,(d,1)} \\ \mathbf{G}_{i,(1,2)} & \mathbf{G}_{i,(2,2)} & \cdots & \mathbf{G}_{i,(d,2)} \\ \vdots & \vdots & \ddots & \cdots \\ \mathbf{G}_{i,(1,N)} & \mathbf{G}_{i,(2,N)} & \cdots & \mathbf{G}_{i,(d,N)} \end{pmatrix} \begin{pmatrix} \mathbf{v}_{i,1} \\ \mathbf{v}_{i,2} \\ \vdots \\ \mathbf{v}_{i,d} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{i=1}^{N} \sum_{j=1}^{d} \mathbf{G}_{i,(j,1)} \mathbf{v}_{i,j} \\ \sum_{i=1}^{N} \sum_{j=1}^{d} \mathbf{G}_{i,(j,2)} \mathbf{v}_{i,j} \\ \vdots \\ \sum_{i=1}^{N} \sum_{j=1}^{d} \mathbf{G}_{i,(j,N)} \mathbf{v}_{i,j} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{i=1}^{N} \mathbf{G}_{i,(1,1)} \mathbf{v}_{i,1} + \sum_{i=1}^{N} \mathbf{G}_{i,(2,1)} \mathbf{v}_{i,2} + \cdots + \sum_{i=1}^{N} \mathbf{G}_{i,(d,1)} \mathbf{v}_{i,d} \\ \sum_{i=1}^{N} \mathbf{G}_{i,(1,2)} \mathbf{v}_{i,1} + \sum_{i=1}^{N} \mathbf{G}_{i,(2,2)} \mathbf{v}_{i,2} + \cdots + \sum_{i=1}^{n} \mathbf{G}_{i,(d,2)} \mathbf{v}_{i,d} \\ \vdots \\ \sum_{i=1}^{N} \mathbf{G}_{i,(1,N)} \mathbf{v}_{i,1} + \sum_{i=1}^{N} \mathbf{G}_{i,(2,N)} \mathbf{v}_{i,2} + \cdots + \sum_{i=1}^{N} \mathbf{G}_{i,(d,N)} \mathbf{v}_{i,d} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{i=1}^{N} \mathbf{G}_{i,(1,1)} \mathbf{v}_{i,1} \\ \sum_{i=1}^{N} \mathbf{G}_{i,(1,2)} \mathbf{v}_{i,1} \\ \vdots \\ \sum_{i=1}^{N} \mathbf{G}_{i,(1,N)} \mathbf{v}_{i,1} \end{pmatrix} + \cdots + \begin{pmatrix} \sum_{i=1}^{N} \mathbf{G}_{i,(d,1)} \mathbf{v}_{i,d} \\ \sum_{i=1}^{N} \mathbf{G}_{i,(d,2)} \mathbf{v}_{i,d} \\ \vdots \\ \sum_{i=1}^{N} \mathbf{G}_{i,(d,N)} \mathbf{v}_{i,d} \end{pmatrix}
\end{aligned} \tag{72}
$$

where $\mathcal{G}_{i,(a,b)}$ refers to the element in row a and column b. The index 'a' refers to the dimension, and 'b' refers to the position in an image.

Recall that $\mathbf{G}_i\mathbf{I}$ in §4.2 gives the Gabor features $\mathbf{g}_i$ at position i in the input image $\mathbf{I}$, and the Gabor features can be calculated by FFT, comparing the matrix expansion of $\mathbf{G}_i\mathbf{I}$ with each column of Eqn. (72), we find that each column in Eqn. (72) also gives a filtered image by a rotated Gabor filter. Figure 35 illustrated this procedure in details. The input image of the convolution is formed in this way: we collect the first dimension of each vector $\mathbf{v}_i, i = 1, \cdots, N$, which gives us a long vector of size N. We rearrange this long vector into an image of size $\sqrt{N} \times \sqrt{N}$, making sure if we concatenating this reformed image row by row, the long vector can be recovered. Another change is the input Gabor

filter is a version of rotate the original ith Gabor filter by 180 degrees. The computation cost of this FFT is $\mathcal{O}(\sqrt{N}^2 \log \sqrt{N}) = \mathcal{O}(N \log N)$. After doing d times this operation, we obtained d vectors of size N. The sum of those N vectors is the result of $\sum_{i=1}^{N} \mathbf{G}_i^T \mathbf{v}_i$.

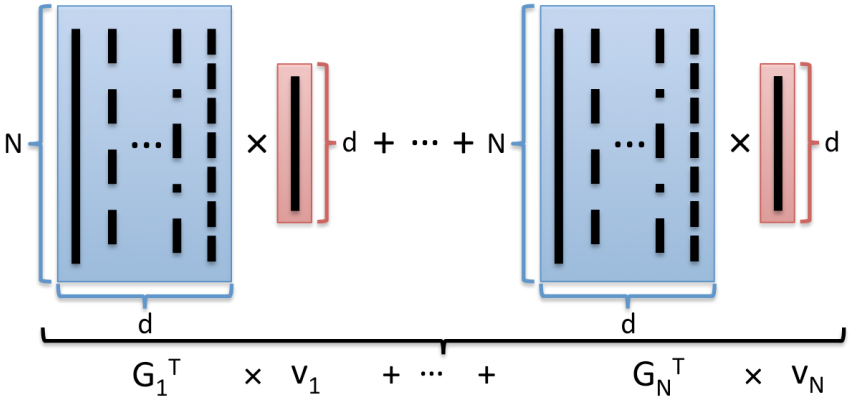
In summary, the fast evaluation procedure is the following:

Algorithm 20 Fast Summation of $\sum_{i=1}^{N} \mathbf{G}_i \mathbf{v}_i$

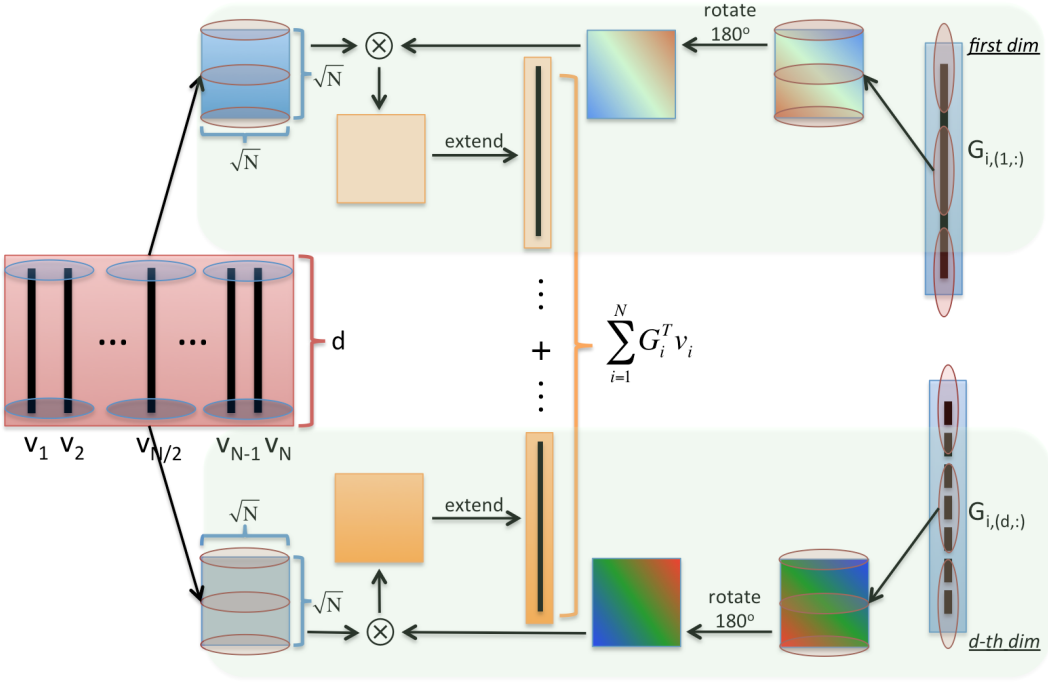
1: **for** $j = 1 : d$ **do**
2: collect all $\{\mathbf{v}_{i,j}\}_{i=1}^{N}$ into one vector $\mathbf{w}$
3: rearrange $\mathbf{w}$ into a 2D matrix $\mathbf{W}$ by continuously padding every $\sqrt{N}$ elements of $\mathbf{w}$ in a row of $\mathbf{W}$.
4: rotate the jth Gabor filter by 180 degrees.
5: convolve the rotated Gabor filter and the derived image $\mathbf{W}$ by FFT
6: concatenate the convolved results into a long vector $\mathbf{w}_j^{fil}$.
7: **end for**
8: $\sum_{i=1}^{N} \mathbf{G}_i \mathbf{v}_i = \sum_{j=1}^{d} \mathbf{w}_j^{fil}$

5.3.4 Parallel Implementation

The most computational expensive part to solve the optimization in Eqn. (**??**) is the evaluation of gradient and the Hessian. Since the Hessian matrix is a full matrix of size $N \times N$, direct calculation according to Eqn. (71) takes $\mathcal{O}(N^4 d^3 + N^3 d^5)$. In fact, in TRON, we do not need to evaluate the exact Hessian matrix every time. The only thing TRON requires is the Hessian-vector multiplication. This leads to a matrix free optimization.



(a) $\sum_{i=1}^{N} \mathbf{G}_i^T \mathbf{v}_i$



(b) fast summation by FFT

Figure 35: *Illustration of fast calculation of* $\sum_{i=1}^{N} \mathbf{G}_i^T \mathbf{v}_i$ The basic idea is rotate the original Gabor filter by 180 degree and reform the input vector into images, then sum over all the convolutions of the rotated Gabor filters and the reformed input images.

5.4 Study of the prior

In this section, we focus on the performance of the prior on some simple examples, and some numerical characteristic of the Hessian matrix as well as the convergence of optimizing Eqn. (62) by steepest descent, which only using the first order derivative and TRON that requires the second order derivative as well.

5.4.1 Learning Ability of the Prior

We first show the ability of our new prior to refine shapes based on training samples. The optimization problem is different from Eqn. (62). We do not utilize any likelihood information, in other words, the optimization problem we try to solve here is

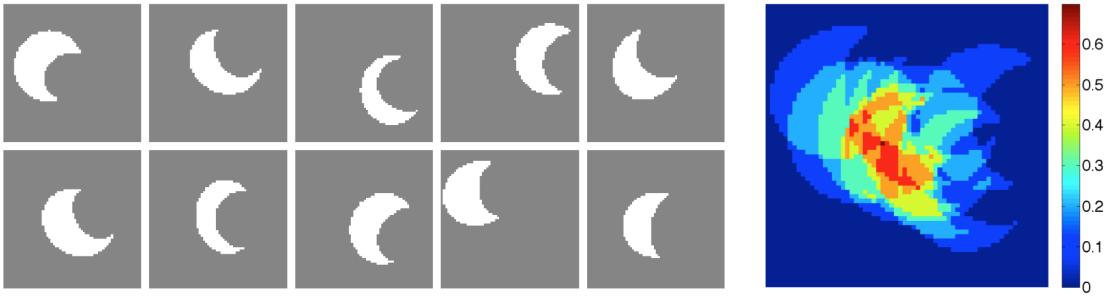
$$\begin{aligned} \boldsymbol{\phi}^* = \quad & \text{argmin}_{\boldsymbol{\phi}} \quad \sum_{i=1}^{N} \boldsymbol{\phi}_i \log p_i^{obj}(\boldsymbol{\phi}) + \sum_{i=1}^{N} (1-\boldsymbol{\phi}_i) \log p_i^{bg}(\boldsymbol{\phi}) \\ & s.t. \qquad 0 \leq \boldsymbol{\phi}_i \leq 1 \end{aligned} \tag{73}$$

In this easy example, we have two sets of training images. One set has images with only one object on it, the other has two objects. All the objects have variations on the shape, and all of them are also rotated and translated to make them do not locate in the same center.

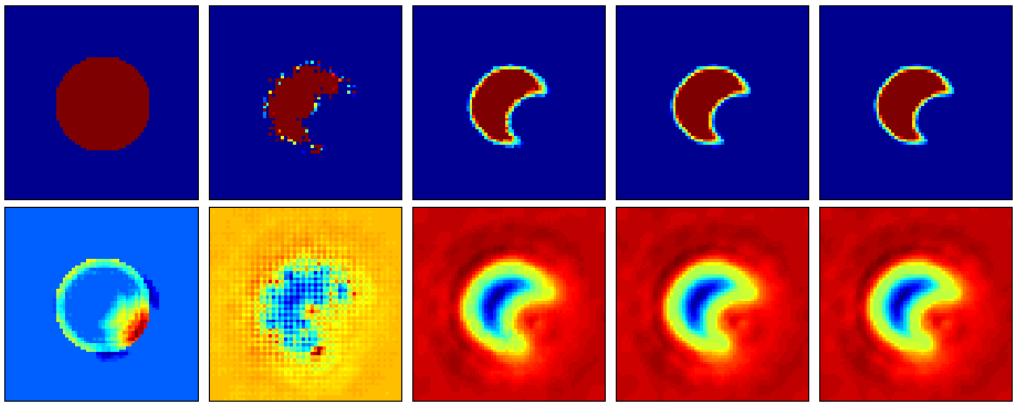
Figure 36 shows the ten training images and the first five iteration results using one training images and ten training images. In Figure 36 (a), the right part shows the mean shape based on all the ten images without any registration. It is obvious that the mean shape cannot give any useful information about the information of the segmented object. Figure 36 (b) shows the first five iterations in TRON using only the first training image. The first row is the updated $\boldsymbol{\phi}$ in each iteration, the second row gives the corresponding gradient based on the updated $\boldsymbol{\phi}$. We can see that after 3 iterations, the input circle has been refined to similar to the first shape. As the TRON iterates, the boundary becomes more clear. Figure 36 (c) shows the first five iterations using 10 training images. After 3 iterations, the shape becomes like a moon shape but does not look like any special one of those 10 images. It roughly looks like a mean shape. Compared to the mean shape in

Figure 36 (a), which gives no sense, even without registration, our prior can still refine the input shape into the shapes constrained by the training images.

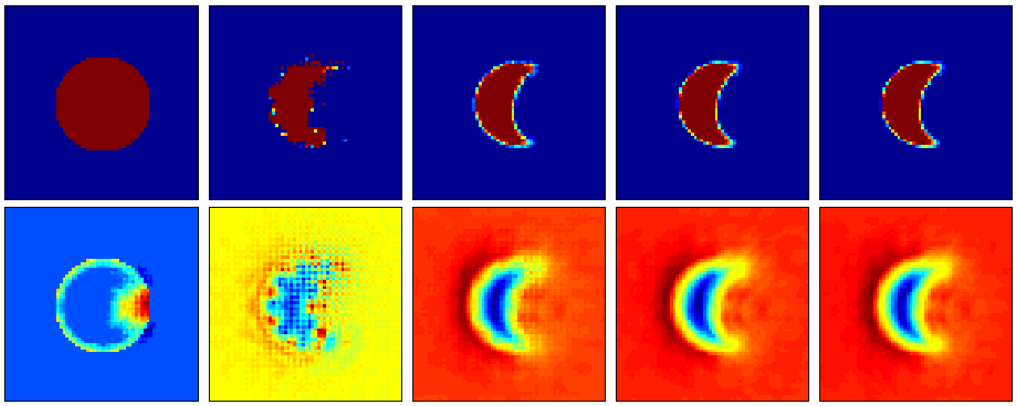
To make the example a little harder, Figure 37 tests our prior on a two-objects case. Similarly, we use one and ten training images respectively. As Figure 37 (b) shows, even the initial input and the training image (the first one) have different configuration, where the initial only have one objects, and the initial input is on the center of image, however those two objects on the training image locate on the right bottom corner, after the first iteration, the output starts to converge to the shapes of the training image, although the updated output locates the same as the initial guess. The ten training images case is a little harder, but after four iterations, the two objects become separated, and in the fifth iteration, the wall become those two objects becomes more clear. And the shape converges to the mean shape without any registration as well.



(a) training images

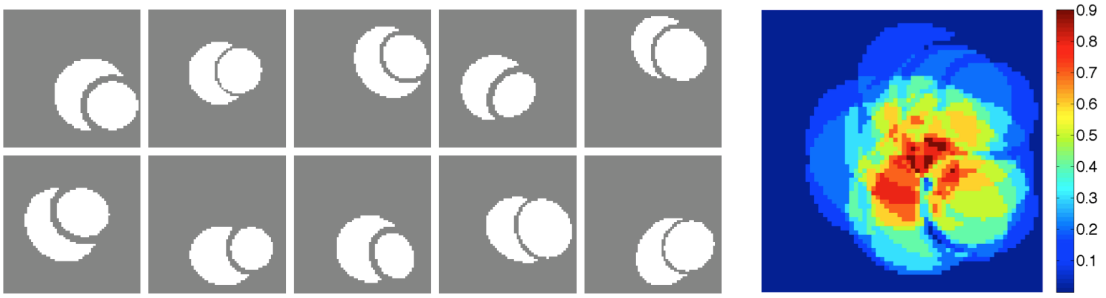


(b) first five iterations using **only the first one** training image

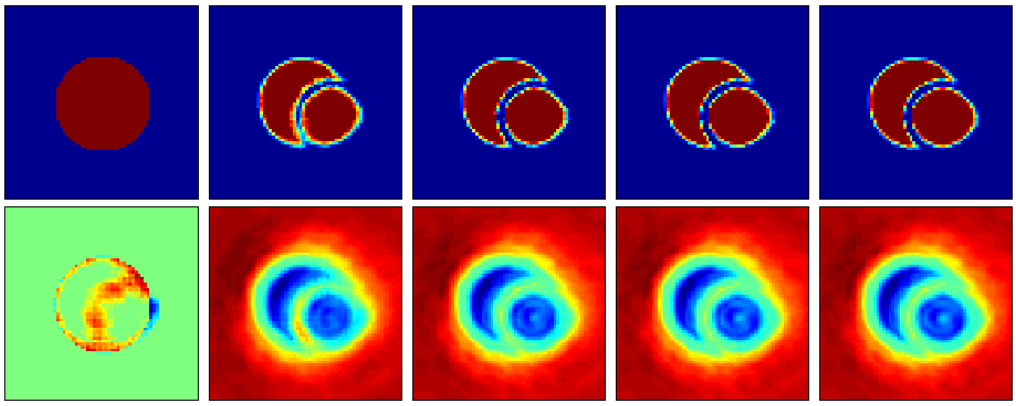


(c) first five iterations using **all ten** training images

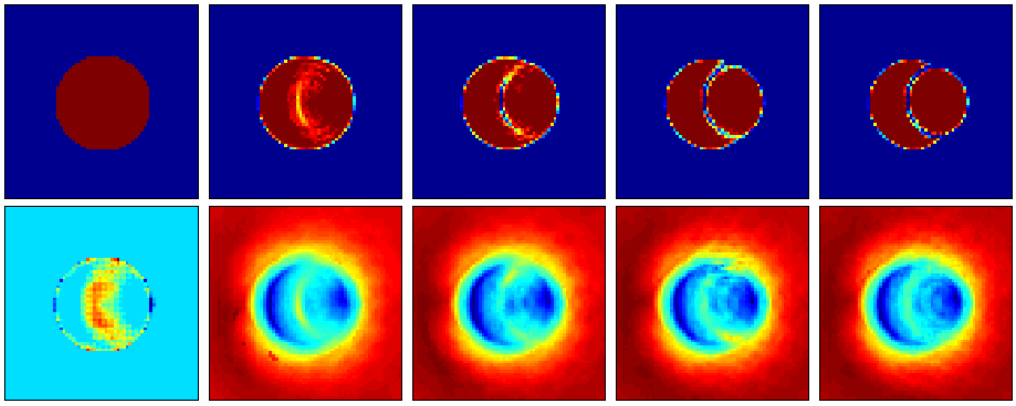
Figure 36: *First five iterations of the one object case.* (a) shows all the ten images used in this example to learn the prior. In this case, there is only one object in the image with different shapes, rotation and translation; (b) gives the first five iterations using only the first one image as training samples. The first row is the update of the solutions, and the second row shows the corresponding gradient evaluated based on the given input; (c) gives the first five iterations using all the ten training images. The first row is the update of the solutions, and the second row shows the corresponding gradient evaluated based on the given input.



(a) training images



(b) first five iterations using **only the first one** training image



(c) first five iterations using **all ten** training images

Figure 37: *First five iterations of the two objects case.* (a) shows all the ten images used in this example to learn the prior. In this case, there are two objects in the image with different shapes, rotation and translation; (b) gives the first five iterations using only the first one image as training samples. The first row is the update of the solutions, and the second row shows the corresponding gradient evaluated based on the given input; (c) gives the first five iterations using all the ten training images. The first row is the update of the solutions, and the second row shows the corresponding gradient evaluated based on the given input.

5.4.2 Comparison between steepest descent and TRON

To demonstrate evaluation the Hessian is necessary to accelerate the converging of optimization, we solved one example problem in Eqn. (62) through two different approaches: steepest descent and TRON [84]. Steepest descent only need the first order derivative (gradient) to compute the updating direction; TRON also calculates the second order derivative (Hessian) to make the optimization converge to a local or global optima faster. Figure 38 gives the convergence result of solving the one-object example using 1, 10, 100 training images respectively by both steepest descend and TRON. The convergence is measured by the norm of the gradient. Ideally, once the global optima was found, there is no more update and the norm of gradient should be zero. From Figure 38, It is clear that after 3 iterations, TRON using 1, 10, 100 training images all nearly converges to the optimal solution. However, the steepest descent converges much slower and need much more iterations to reach the same error.

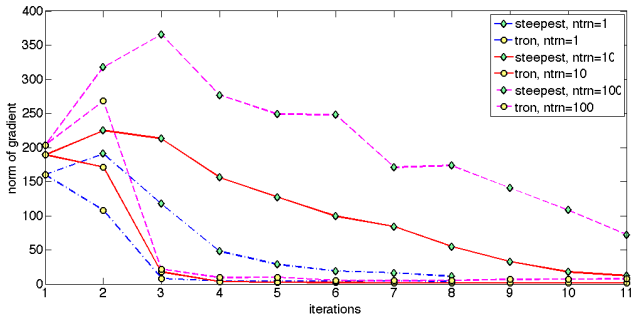


Figure 38: *Convergence comparison between steepest descent and TRON.* We solved the one object problem using 1, 10, 100 training images respectively by two optimization approaches: steepest descent and TRON. The 'diamond' marker indicates the iterations of steepest decent, and the 'circle' marker indicates the iterations of TRON. The convergence is measured by the norm the gradient.

5.5 Experimental Results

In this section, we demonstrate the performance of our prior by two kinds of experiments. The first one is the highly incomplete least square inverse problem, which is ill-posed and standard regularization fails. In this case, our prior as a new regularizer could help to recover the original data. The second one is using the new prior for medical image segmentation, which help improve the accuracy of pure likelihood function.

5.5.1 Least Square Inverse Problem

In this section, we consider a simple linear forward model,

$$\mathbf{y} = f(\boldsymbol{\phi}) = \mathbf{A}\boldsymbol{\phi}, \;\; \mathbf{I} = \mathbf{A}\boldsymbol{\phi} + \varepsilon \tag{74}$$

The least square problem is defined as: given $\mathbf{y}$ and the generation operator $\mathbf{A}$, find the original $\boldsymbol{\phi}$.

In the case that $\mathbf{A}$ is full rank, the solution is straight forward $\boldsymbol{\phi} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\mathbf{y}$. Unfortunately, usually the operator is high singular such that $\mathbf{A}^T\mathbf{A}$ is non-invertible. People often add one more term as regularizer such as Tychonov regularization, smoothness, Total Variation (TV), etc.. We consider this inverse problem by adding our new prior model as a regularizer, and compare our result with several popular regularization terms.

This input sources $\boldsymbol{\phi}$ we used in our experiments are the same as in Figure 36 and Figure 37, i.e., the one object and two objects images. The operator $\mathbf{A}$ is generated by $\mathbf{A} = \mathbf{B}^T\mathbf{B}$ where $\mathbf{B}$ is a random matrix of size $M \times N$. To make $\mathbf{A}$ singular, we set to be M much less than N. In our case, we use $M = 0.05N$ and $M = 0.1N$ respectively. Besides, we also add 50% noise to make this problem a little harder.

Figure 39 shows the result of the case $\mathbf{A}$ has 5% observation (5% eigenvalues are non-zeros) and without noise. The first image is the real $\boldsymbol{\phi}$, and the second one gives the generated image. The third one gives a typical lease square solution with a small regularizer to make $\mathbf{A}$ invertible. It can be seen that the mean square error (mse) between the recovered $\boldsymbol{\phi}'$ and the real $\boldsymbol{\phi}$ is 17.90, which is super high, and the recovered one is almost noise. The fourth image use the same least square object function but with constraints that all $\boldsymbol{\phi}_i$ should in the range $[0, 1]$. It still cannot recover the original image. The fifth one try to solve the least square objection with the Gaussian prior, which is in essence the mean shape refinement. Due to the training images are rotated and translated, similar to Figure 36 (a), the mean shape cannot gives any useful information if no preprocessing (registration) was performed. The sixth image is using our prior, the mse is much lower compared with other prior, and it is clear that the image was recovered very well. The 7th image is using the smooth prior, which basically is $\boldsymbol{\phi}^T\mathbf{S}\boldsymbol{\phi}$, where $\mathbf{S}$ is a operator try to averaging each pixel by nearby 4 pixels. It can somehow recover the original image. However compared with our prior, the recovered image is highly blurred and it has some other clutters. The last two images shows the using the combination of smooth prior with Gaussian prior and our prior respectively. The smooth prior can help to improve our prior a little bit, but not significantly.

Figure 40 applies the same operator $\mathbf{A}$ but with 50% noise. Similarly, our prior outperforms all the other priors. Even the images is corrupted seriously by white noise, our prior can still recover the original $\boldsymbol{\phi}$ very well. Figure 41 and Figure 41 shows the resulting using the two objects case using 5% observed $\mathbf{A}$ with or without noise. In the two objects case,

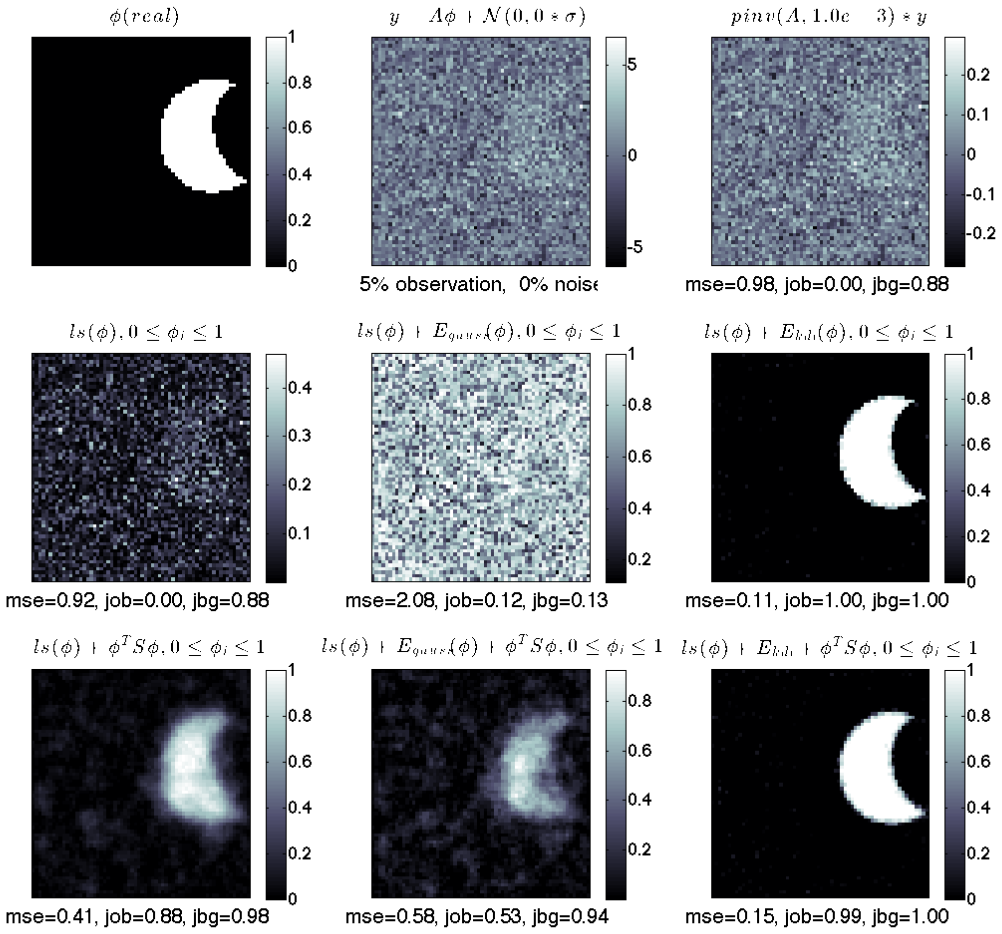


Figure 39: *Incomplete least square of one object case with* 5% *observation and* 0% *noise.*

our prior can still recover the original source ϕ. The wall between two objects is much clearer than the smooth prior, even if there is huge noise.

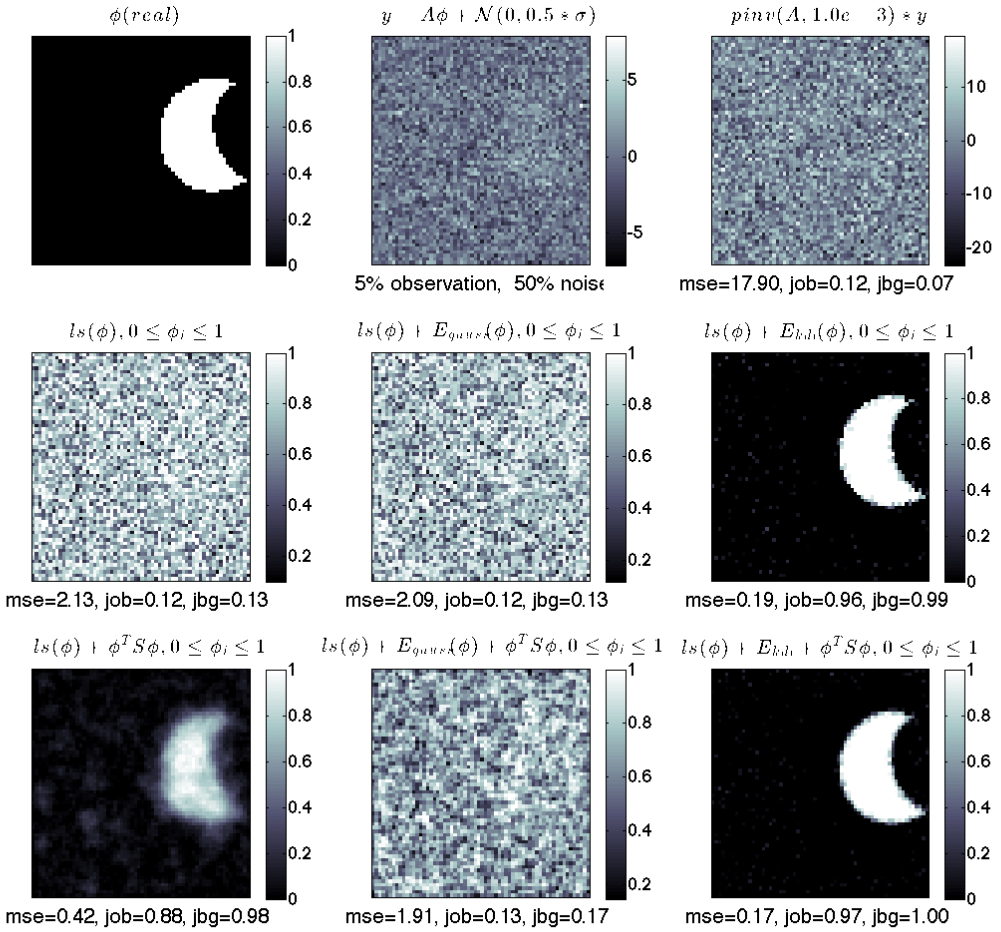


Figure 40: *Incomplete least square of one object case with* 5% *observation and* 50% *noise.*

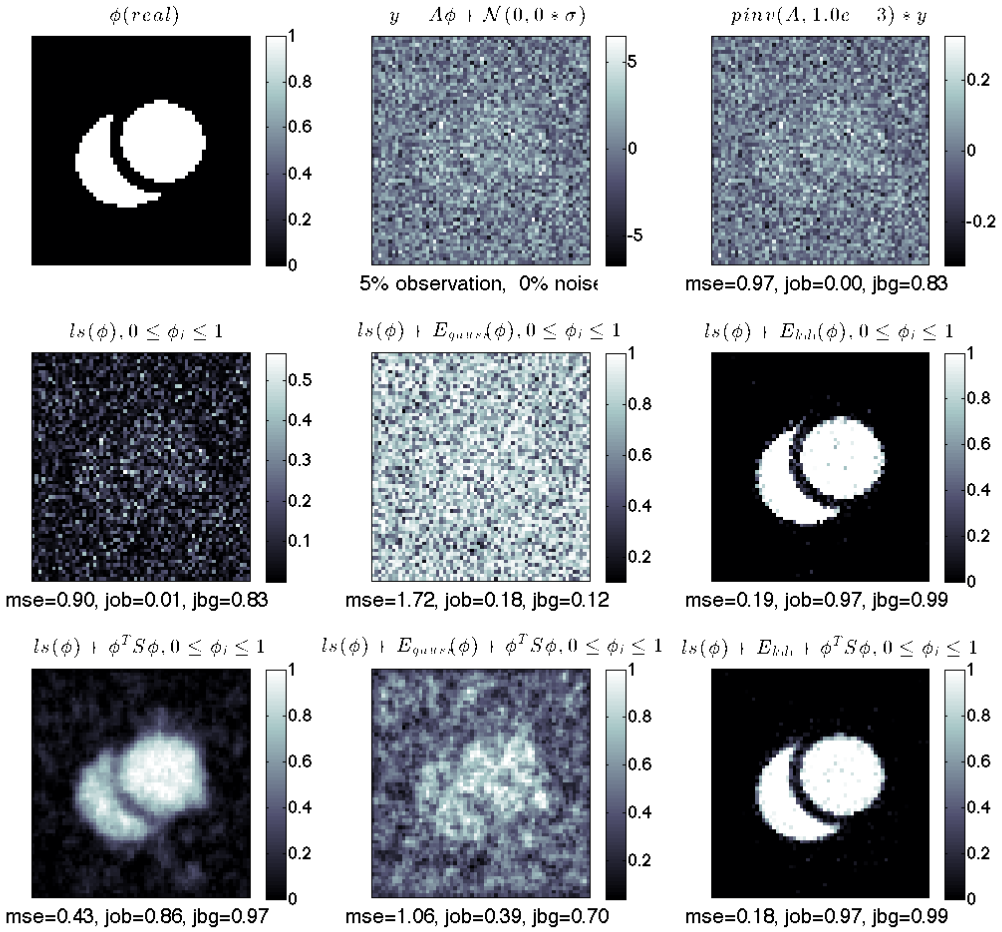


Figure 41: *Incomplete least square of two objects case with* 5% *observation and* 0% *noise.*

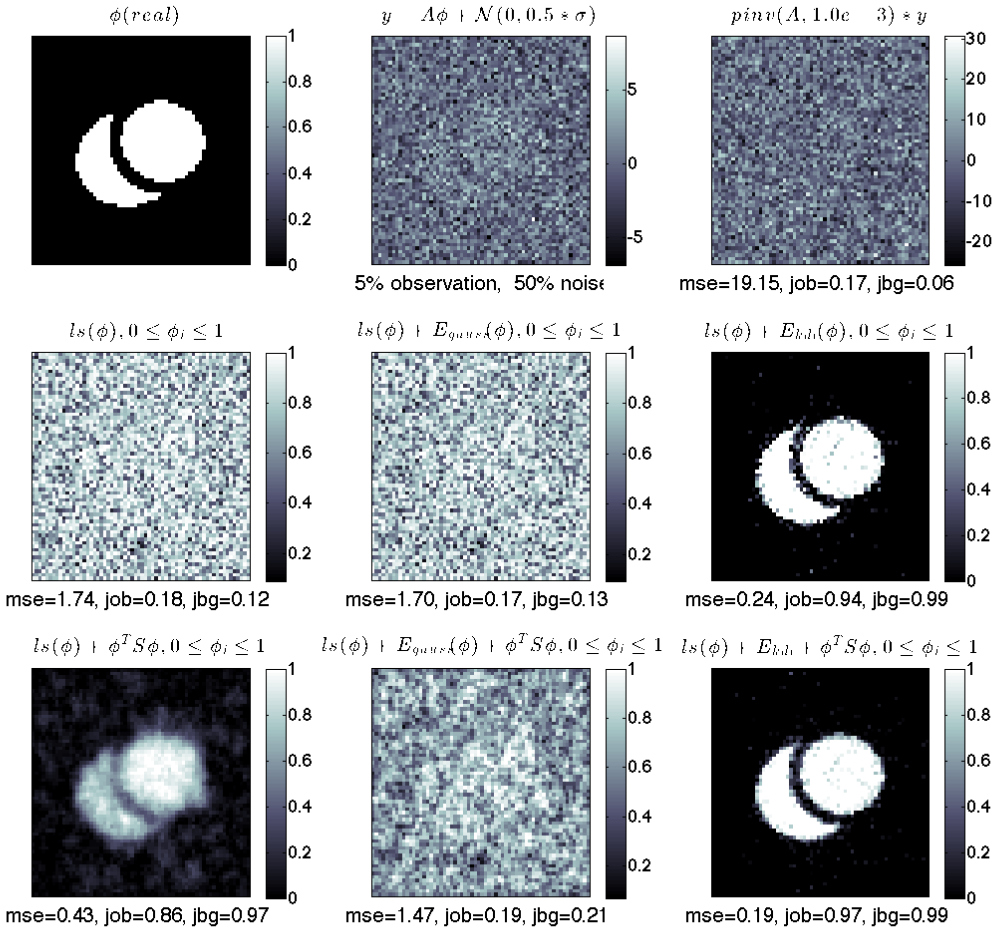


Figure 42: *Incomplete least square of two objects case with* 5% *observation and* 50% *noise.*

5.5.2 Medical Image Segmentation

In this section, we use our new prior in the Bayesian framework in Eqn. (47). The likelihood was introduced in chapter 4. Table 26 gives the accuracy combining both the likelihood in Tabel 20 and our new prior. Generally speaking, there is no significance between the segmentation results between using the likelihood only and the the combination of the likelihood and the prior in terms of the Jaccard index. However, if we look at the exact segmentation, we would found almost all the results with our prior have been refined from the likelihood segmentation.

Figure 43 gives some examples that our prior improves the segmentation. For example, even with high noise, the likelihood segmentations have lots of clutters, our prior can remove those clutters. In some other cases, say the 8th figure, the likelihood does not give a connected object; some position are separated into small spots. Our new prior can connect all those small pieces into one object. Figure 44 shows some cases that our prior does not work that well. It does not mean the new prior does not work. In fact our prior can also remove lots of clutters and artifacts which the likelihood misclassifies. However, because the prior is statistical, it might add some 'ghost' pixels, which are mainly along the boundary, in the final segmentation, make it look more similar to a certain shape in our training examples. But we think even our prior has this drawback, it is still useful because deleting artifacts even in highly noisy images is very helpful.

Table 26: *Segmentation accuracy using our prior*: This table gives the segmentation accuracy using our prior. The likelihood term is the same as Table 20.

noise \ #trn	J^{ob}				J^{bg}			
	1	*10*	*100*	*1000*	*1*	*10*	*100*	*1000*
0%	0.5839	0.7799	0.8832	0.9152	0.9441	0.9729	0.9880	0.9918
10%	0.6481	0.7599	0.8513	0.8842	0.9621	0.9687	0.9838	0.9882
20%	0.5148	0.7336	0.8333	0.8634	0.9331	0.9649	0.9813	0.9858

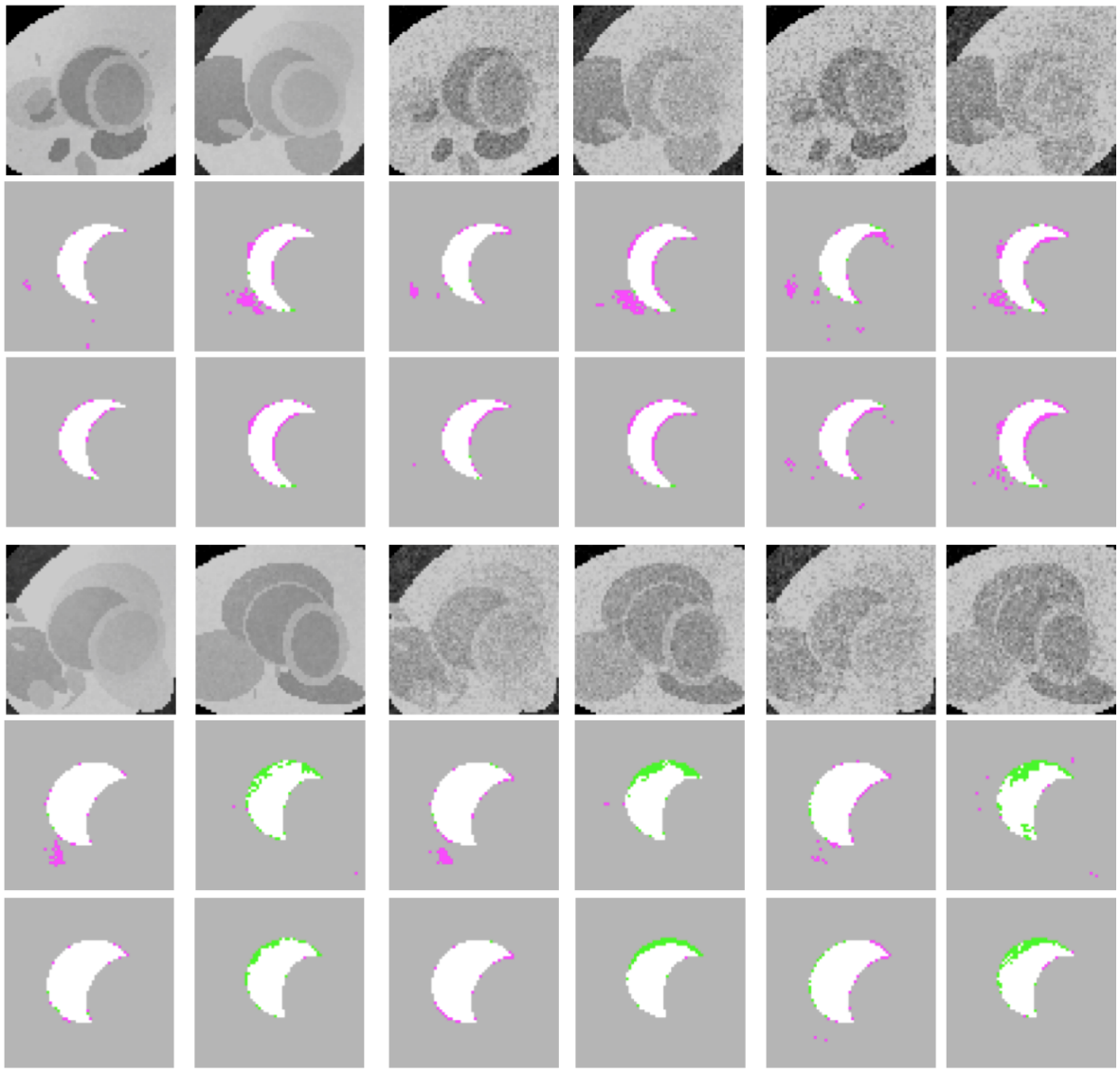


Figure 43: *Examples of segmentation with our new priors* This figure gives some good case that our prior works well. The first two columns are images without noise. The middle two columns are the same images with 10% noise. The last two columns are of 20% noise. The second and the fifth rows are the segmentation of pure likelihood; the third and the sixth row are the segmentation of likelihood with our prior. To indicate the difference between our segmentations and the true segmentations, we use '**white**' indicates the overlap region between the true label and our segmentation, '**green**' indicates the left region of the true label excluding the overlap, and '**magenta**' represents the remaining region of our segmentation besides the overlap.

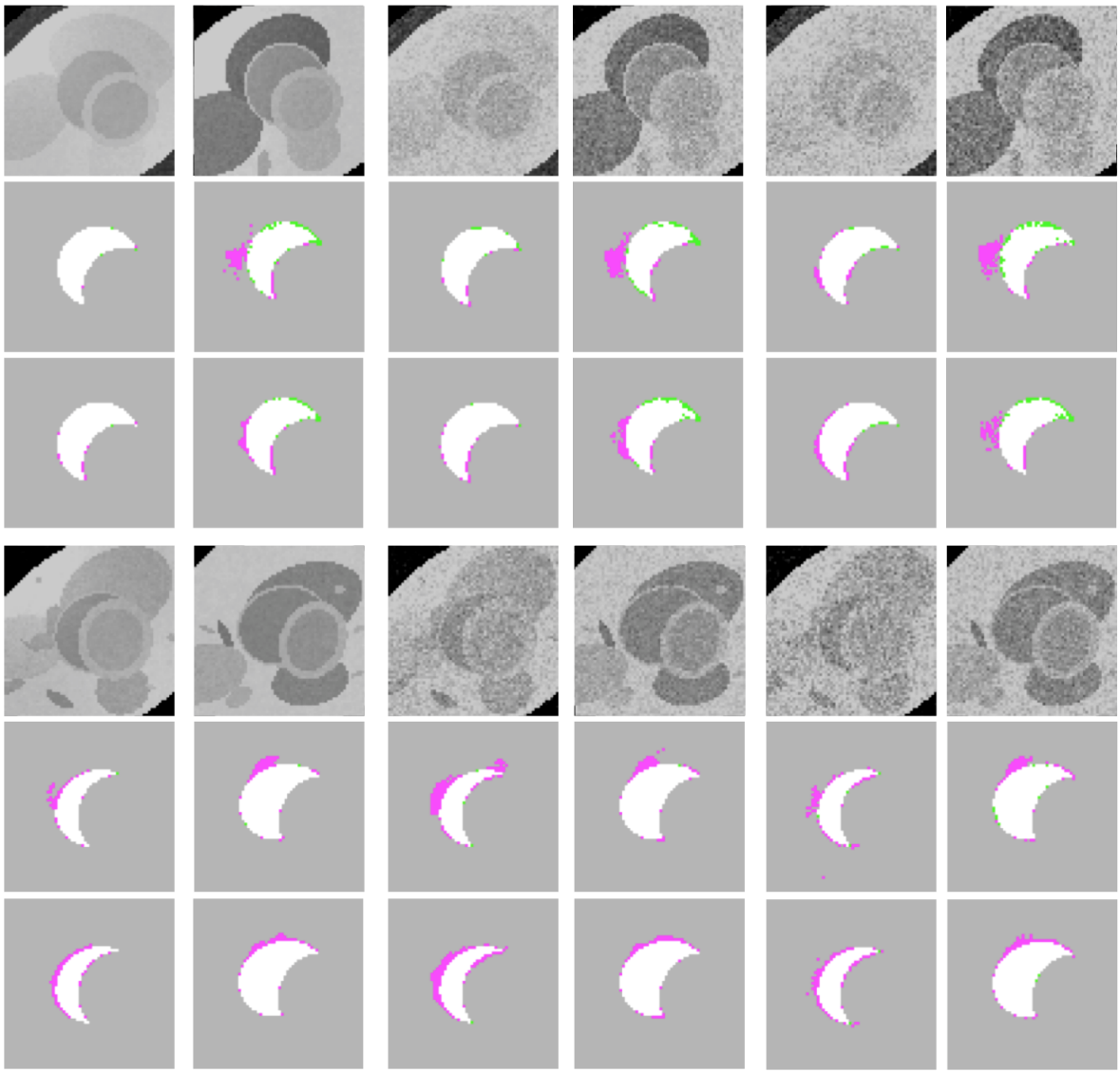


Figure 44: *Examples of segmentation with our new priors* This figure gives some cases that our prior does not work well. The first two columns are images without noise. The middle two columns are the same images with 10% noise. The last two columns are of 20% noise. The second and the fifth rows are the segmentation of pure likelihood; the third and the sixth row are the segmentation of likelihood with our prior. To indicate the difference between our segmentations and the true segmentations, we use '**white**' indicates the overlap region between the true label and our segmentation, '**green**' indicates the left region of the true label excluding the overlap, and '**magenta**' represents the remaining region of our segmentation besides the overlap.

CHAPTER VI

CONCLUSION

This research consists of new methods for medical image segmentation and computational geometry problems. In particular, we proposed new techniques and developed a set of basic tools built on top of MPI and OpenMP for massively parallel nearest neighbors search and kernel independent summation, and scales them to thousands of cores. We also introduce new shape prior that can be incorporated with MAP estimation as well as other inverse problems like image denoising.

We briefly summarize the contributions of this dissertation:

- Parallel nearest neighbor searches. We presented what we believe to be the first attempt at solving KNN problem in both high dimensions and on supercomputing platforms. Our RKD TREE is a novel distributed memory spatial indexing structure which is capable of accelerating queries for certain high dimensional data sets and.

- Parallel kernel summation. We proposed a new scheme for high dimensional N-body problems, which is based only on kernel evaluations. It uses neighbor-based pruning, and uses neighbor-sampled interpolative decomposition to approximate the far field.

- Bayesian medical image segmentation with new statistical shape prior. We investigated a new shape prior which based on KDE in high dimensions. Our new prior requires no registration of training samples as preprocessing, and can be easily combined with other inverse problem as a regularization term.

Considerations for the future research include: (1) improving both the absolute running time and the scalability of our KNN code by (a) reducing the communication cost of RKD TREE, (b) porting our code to modern heterogeneous systems, (c) using other approximate

and randomized pruning measures. (2) deriving a rigorous error bound that incorporates our sampling scheme in ASKIT. (b) optimizing ASKITin adaptive determination of the skeleton size, and improving the sampling. (c) notice that our scheme only works for Euclidean distance for this time being, we might extend it to non-metric spaces. (3) extending our shape prior to image registration and fully tuning the optimization solving codes.

REFERENCES

[1] "Mathematics of analysis of petascale data," tech. rep., DOE Office of Science, 2008.

[2] "The opportunities and challenges of exascale computing," tech. rep., DOE Office of Science, 2010.

[3] AL-FURAJH, I., ALURU, S., GOIL, S., and RANKA, S., "Parallel construction of multidimensional binary search trees," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 11, no. 2, pp. 136–148, 2000.

[4] ANDONI, A. and INDYK, P., "E2LSH 0.1 User manual," 2005. `http://www.mit.edu/~andoni/LSH`.

[5] ANDONI, A. and INDYK, P., "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *COMMUNICATIONS OF THE ACM*, vol. 51, no. 1, p. 117, 2008.

[6] ANDREWS, S., MCINTOSH, C., and HAMARNEH, G., "Convex multi-region probabilistic segmentation with shape prior in the isometric log-ratio transformation space," in *IEEE International Conference on Computer Vision*, pp. 2096–2103, 2011.

[7] BACHE, K. and LICHMAN, M., "UCI machine learning repository," 2013.

[8] BAGON, S., "Matlab wrapper for graph cut," December 2006.

[9] BELONGIE, S., MALIK, J., and PUZICHA, J., "Shape matching and object recognition using shape contexts," *PAMI*, pp. 509–522, 2002.

[10] BERCHTOLD, S., KEIM, D., and KRIEGEL, H., "The X-tree: An index structure for high-dimensional data," *Readings in multimedia computing and networking*, vol. 12, p. 451, 2002.

[11] BEYGELZIMER, A., KAKADE, S., and LANGFORD, J., "Cover trees for nearest neighbor," in *ICML*, pp. 97–104, 2006.

[12] BISHIP, C. M., *Pattern Recognition and Machine Learning*. Springer, 2007.

[13] BOWMAN, A. W., "An alternative method of cross-validation for the smoothing of kernel density estimates," *Biometrika*, vol. 71, pp. 353–360, 1984.

[14] BOYKOV, Y. and JOLLY, M., "Interactive graph cuts for optimal boundary and region segmentation of objects in n-d images," in *IEEE International Conference on Computer Vision*, pp. 105–112, 2001.

[15] BOYKOV, Y. and KOLMOGOROV, V., "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 1124–1137, 2004.

[16] CASELLES, V., KIMMEL, R., and SHAPIRO, G., "Geodesic active contours," *Int. J. of Computer Vision*, vol. 22, no. 1, pp. 61–79, 1997.

[17] CHAN, T. and VESE, L., "Active contours without edges," *IEEE Transactions on Image Processing*, vol. 10, pp. 266–277, 2001.

[18] CHAN, T. and ZHU, W., "Level set based shape prior segmentation," *CVPR*, pp. 1063–6919, 2005.

[19] CHANG, C. Y. and CHUNG, P. C., "Medical image segmentation using a contextual-constraint-based hopfield neural cube," *Image and Vision Computing*, vol. 19, no. 9-10, pp. 669–678, 2001.

[20] CHARPIAT, G., FAUGERAS, O., and KERIVEN, R., "Shape statistics for image segmentation with prior," *CVPR*, pp. 1–6, 2007.

[21] CHEN, S., CREMERS, D., and RADKE, R. J., "Image segmentation with one shape prior - a template-based formulation," *Image and Vision Computing*, vol. 30, pp. 1032–1042, 2012.

[22] CHENG, H., GREENGARD, L., and ROKHLIN, V., "A fast adaptive multipole algorithm in three dimensions," *Journal of Computational Physics*, vol. 155, pp. 468–498, 1999.

[23] CHUANG, Y. Y., CURLESS, B., SALESIN, D. H., and SZELISKI, R., "A bayesian approach to digital matting," in *IEEE International Conference on Computer Vision*, pp. 264–271, 2001.

[24] CIACCIA, P., PATELLA, M., and ZEZULA, P., "M-tree an efficient access method for similarity search in metric spaces," *Proceedings of the 23rd VLDB Conference*, pp. 426–435, 1997.

[25] COMANICIU, D. and MEER, P., "Meah shift: A robust appraoch toward feature space analysis," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 603–619, 2002.

[26] COMANICIU, D. and MEER, P., "Mean shift: A robust approach toward feature space analysis," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 603–619, 2002.

[27] COOTES, T., EDWARDS, G. J., and TAYLOR, C. J., "Active appearance models," *PAMI*, pp. 681–685, 2001.

[28] COOTES, T. and TAYLOR, C., "A mixture of model for representing shape variation," *IVC*, pp. 110–119, 1997.

[29] COOTES, T., TAYLOR, C., COOPER, D., and GRAHAM, J., "Active shape model - their training and application," *CVIU*, vol. 61, pp. 38–59, 1995.

[30] COOTES, T. F., TAYLOR, C. J., COOPER, D. H., and GRAHAM, J., "Active shape model - their training and application," *Comput. Vision Image Understand.*, vol. 61, no. 1, pp. 38–59, 1995.

[31] CREMERS, D., KOHLBERGER, T., and SCHNOERR, C., "Shape statistics in kernel space for variational image segmentation," *Pattern Recognition*, vol. 36, pp. 1929–1943, 2003.

[32] CREMERS, D., OSHER, S. J., and SOATTO, S., "Kernel density estimation and intrinsic alignment for shape priors in level set segmentation," *IJCV*, pp. 335–351, 2006.

[33] CREMERS, D., ROUSSON, M., and DERICHE, R., "A review of statistical approaches to level set segmentation: integrating color, texture, motion and shape," *International Journal of Computer Vision*, vol. 72, pp. 195–215, Apr. 2007.

[34] CREMERS, D., SCHMIDT, F. R., and BARTHEL, F., "Shape priors in variational image segmentation: convexity, lipschitz continuity and globally optimal solutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–6, 2008.

[35] CREMERS, D., SCHNOERR, C., and WEICKERT, J., "Diffusion snakes: combining statistical shape knowledge and image information in a variational framework," *IEEE First Workshop on Variational and Level Set Methods*, pp. 137–144, 2001.

[36] CREMERS, D. and SOATTO, S., "A pseudo distance for shape priors in level set segmentation," *IEEE Workshop on Variational, Geometric and Level Set Methods in Computer Visions*, 2003.

[37] CREMERS, D., OSHER, S. J., and SOATTO, S., "Kernel density estimation and intrinsic alignment for shape priors in level set segmentation," *International Journal of Computer Vision*, vol. 69, no. 3, pp. 335–351, 2006.

[38] DASGUPTA, S. and FREUND, Y., "Random projection trees and low dimensional manifolds," in *Proceedings of the 40th annual ACM symposium on Theory of computing*, pp. 537–546, ACM, 2008.

[39] DASGUPTA, S. and FREUND, Y., "Random projection trees and low dimensional manifolds," in *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, 2008.

[40] DATAR, M., IMMORLICA, N., INDYK, P., and MIRROKNI, V., "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*, pp. 253–262, ACM, 2004.

[41] DAUGMAN, J. G., "Complete discrete 2d gabor transforms by neural networks for image analysis and compression," *IEEE Trans. ASSP*, vol. 36, pp. 169–179, 1988.

[42] DEVINE, K. D., BOMAN, E. G., and KARYPIS, G., "Partitioning and load balancing for emerging parallel applications and architectures," in *Frontiers of Scientific Computing* (HEROUX, M., RAGHAVAN, A., and SIMON, H., eds.), Philadelphia: SIAM, 2006.

[43] DHILLON, I. and MODHA, D., "A data-clustering algorithm on distributed memory multiprocessors," *Large-Scale Parallel Data Mining*, pp. 802–802, 2000.

[44] DUIN, R. P. W., "On the choice of smoothing parameters of parzen estimators of probability density functions," *IEEE Transactions on Computers*, vol. 25, pp. 1175–1179, 1976.

[45] ESSAFI, S., LANGS, G., and PARAGIOS, N., "Left ventricle segmentation using diffusion wavelets and boosting," in *Medical Image Computing and Computer Assisted Intervention, MICCAI*, pp. 919–926, 2009.

[46] ET AL., K. S., "Sparse decomposition and modeling of anatomical shape variation," *IEEE Transaction on Medical Imaging*, pp. 1625–1635, 2007.

[47] ETYNGIER, P., SEGONNE, F., and KERIVEN, R., "Shape priors using manifold learning techniques," *ICCV*, pp. 1–8, 2007.

[48] FAN, R. E., CHANG, K. W., HSIEH, C. J., WANG, X. R., and LIN, C. J., "Liblinear: A library for large linear classification," *Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[49] FERRARI, V., JURIE, F., and SCHMID, C., "Accurate object detection with deformable shape models learnt from images," *CVPR*, pp. 1–8, 2007.

[50] FREEDMAN, D. and ZHANG, T., "Interative graph cut based segmentation with shape priors," *IEEE Conf. on Computer Vision and Pattern Recognition*, pp. 20–25, 2005.

[51] FREIDMAN, J. H., BENTLEY, J. L., and FINKEL, R. A., "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, pp. 209–226, 1977.

[52] FREUND, Y., DASGUPTA, S., KABRA, M., and VERMA, N., "Learning the structure of manifolds using random projections," in *Neural Information Processing Systems (NIPS)*, 2007.

[53] FUKUNAGA, K. and NARENDRA, P. M., "A brach and bound algorithm for computing k-nearest neighbors," *IEEE Trans. Comput.*, vol. 24, pp. 750–753, 1975.

[54] FULKERSON, B., VEDALDI, A., and SOATTO, S., "Class segmentation and object localization with superpixel neighborhoods," *IEEE International Conference on Computer Vision*, pp. 670–677, 2009.

[55] GANAPATHYSUBRAMANIAN, B. and ZABARAS, N., "A non-linear dimension reduction methodology for generating data-driven stochastic input models," *Journal of Computational Physics*, vol. 227, no. 13, pp. 6612–6637, 2008.

[56] GARCIA, V., DEBREUVE, E., and BARLAUD, M., "Fast k nearest neighbor search using GPU," 2008.

[57] GEMAN, S. and GEMAN, D.

[58] GIMBUTAS, Z. and ROKHLIN, F., "A generalized fast mulipole method for nonoscillatory kernels," *SIAM Journal on Scientific Computing*, vol. 24, no. 3, pp. 796–817, 2002.

[59] GRAY, A. and MOORE, A., "N-Body'problems in statistical learning," *Advances in neural information processing systems*, pp. 521–527, 2001.

[60] GREENGARD, L. and STRAIN, J., "The fast Gauss transform," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 1, pp. 79–94, 1991.

[61] GRIEBEL, M. and WISSEL, D., "Fast approximation of the discrete Gauss transform in higher dimensions," *Journal of Scientific Computing*, vol. 55, no. 1, pp. 149–172, 2013.

[62] GROPP, W., LUSK, E., DOSS, N., and SKJELLUM, A., "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, Sept. 1996.

[63] HAGHANI, P., MICHEL, S., and ABERER, K., "Lsh at large distributed knn search in high dimensions."

[64] HALKO, N., MARTINSSON, P., and TROPP, J., "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, vol. 53, p. 217, 2011.

[65] HOGEA, C., DAVATZIKOS, C., and BIROS, G., "An image driven parameter estimation problem for a reaction-diffusion glioma growth model with mass effects," *Journal of Mathematical Biology*, vol. 56, pp. 793–825, 2008.

[66] JAIN, A., MURTY, M., and FLYNN, P., "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[67] JIANG, T., JURIE, F., and SCHMID, C., "Learning shape prior models for object matching," *CVPR*, pp. 848–855, 2009.

[68] JONES, P. W., OSIPOV, A., and ROKHLIN, V., "A randomized approximate nearest neighbors algorithm," tech. rep., YALEU/DCS/RR-1434, Yale University, New Haven, CT, 2010.

[69] JOSHI, M., "Parallel k-means algorithm on distributed memory multiprocessors," *Computer*, vol. 9, 2003.

[70] KANUNGO, T., MOUNT, D., NETANYAHU, N., PIATKO, C., SILVERMAN, R., and WU, A., "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 881–892, 2002.

[71] KAPUR, T., GRIMSON, W. E. L., WELLS, W. M., and KIKINIS, R., "Segmentation of brain tissue from magnetic resonance images," in *Medical Image Analysis*, vol. 1, 1996.

[72] KARGER, D. and RUHL, M., "Finding nearest neighbors in growth restricted metrics," *Proceeding of STOC*, 2002.

[73] KASS, M., WITKIN, A., and D. TERZOPOULOS, JOURNAL = IJCV, T. . S. V. . . P. . . Y. . .

[74] KASS, M., WITKIN, A., and TERZOPOULOS, D., "Snakes: active contour model," *Int. J. of Computer Vision*, vol. 1, pp. 321–331, 1988.

[75] LASHUK, I., CHANDRAMOWLISHWARAN, A., H. LANGSTON, T.-A. N., SAMPATH, R., SHRINGARPURE, A., VUDUC, R., L. YING, D. Z., and BIROS, G., "A massively parallel adaptive fast-multipole method on heterogeneous architectures," in *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2009.

[76] LEE, D., GRAY, A., and MOORE, A., "Dual-tree fast gauss transforms," *Advances in Neural Information Processing Systems*, vol. 18, p. 747, 2006.

[77] LEE, D. and GRAY, A. G., "Fast high-dimensional kernel summations using the monte carlo multipole method.," in *NIPS*, pp. 929–936, 2008.

[78] LEE, D., VUDUC, R., and GRAY, A. G., "A distributed kernel summation framework for general-dimension machine learning," *Proc. SIAM Int'l. Conf. Data Mining (SDM)*, pp. 391–402, 2013.

[79] LEVENTON, M., GRIMSON, W. E., and FAUGERAS, O., "Statistical shape influence in geodesic active contours.," *IEEE Conference on Computer Vision and Pattern Recognition*, p. 13161323, 2000.

[80] LI, C., KAO, C., GORE, J. C., and DING, Z., "Minimization of region-scalable fitting energy for image segmentation," *IEEE Transaction on Image Processing*, vol. 17, no. 10, 2008.

[81] LI, C., XU, C., GUI, C., and FOX, M. D., "Distance regularized level set evolution and its application to image segmentation," *IEEE Trans. on Image Processing*, vol. 19, pp. 3243–3254, 2010.

[82] LI, G., LEI, G., and LIU, T., "Grouping of brain mr images via affinity propagation," in *IEEE International Symposium on Circuits and Systems*, pp. 2425–2428, 2009.

[83] LIBERTY, E., WOOLFE, F., MARTINSSON, P., ROKHLIN, V., and TYGERT, M., "Randomized algorithms for the low-rank approximation of matrices," *Proceedings of the National Academy of Sciences*, vol. 104, no. 51, p. 20167, 2007.

[84] LIN, C. J. and MORE, J. J., "Newton's method for large bound-constrained optimization problems," *SIAM Journal on Optimization*, vol. 9, no. 4, pp. 1100–1127, 1999.

[85] LIU, C., "Gabor-based kernel pca with fractional power polynomial models for face recognition," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 26, pp. 572–581, 2004.

[86] LV, Q., JOSEPHSON, W., WANG, Z., CHARIKAR, M., and LI, K., "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, 2007.

[87] MAHONEY, M. and DRINEAS, P., "Cur matrix decompositions for improved data analysis," *Proceedings of the National Academy of Sciences*, vol. 106, no. 3, p. 697, 2009.

[88] MANJUNATH, B. S. and MA, W. Y., "Texture features for browsing and retrieval of image data," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 18, pp. 837–842, 1996.

[89] MILLER, G., TENG, S., THURSTON, W., and VAVASIS, S., "Separators for sphere-packings and nearest neighbor graphs," *Journal of the ACM (JACM)*, vol. 44, no. 1, pp. 1–29, 1997.

[90] MOHAMED, N. A., AHMED, M. N., and FARAG, A., "Modified fuzzy c-mean in medical image segmentation," in *Internatinoal Conference on Acoustics, speech and signal processing*, pp. 3429–3432, 1999.

[91] MOON, L., LONG, D., JOSHI, S., TRIPATH, V., XIAO, B., and BIROS, G., "Parallel algorithms for clustering and nearest neighbor search problems in high dimensions," in *2011 ACM/IEEE conference on Supercomputing, Poster Session*, (Piscataway, NJ, USA), IEEE Press, 2011. `padas.ices.utexas.edu/static/papers/sc11-knn.pdf`.

[92] MORARIU, V. I., SRINIVASAN, B. V., RAYKAR, V. C., DURAISWAMI, R., and DAVIS, L. S., "Automatic online tuning for fast Gaussian summation.," in *NIPS*, pp. 1113–1120, 2008.

[93] MUJA, M. and LOWE, D. G., "Fast approximate nearest neighbors with automatic algorithm configuration," in *In VISAPP International Conference on Computer Vision Theory and Applications*, pp. 331–340, 2009.

[94] MUMFORD, D. and SHAH, J., "Optimal approximation by piecewise smooth functions and associated variational problems," *Comm. Pure Applied Math*, vol. 42, pp. 577–685, 1989.

[95] MURPHY, P. M. and AHA, D. W., "Uci repository of machine learning databases," *Online*, 1992. http://archive.ics.uci.edu/ml/datasets.

[96] OLSON, C., "Parallel algorithms for hierarchical clustering," *Parallel computing*, vol. 21, no. 8, pp. 1313–1325, 1995.

[97] OMOHUNDRO, S. M., "Efficient algorithms with neural network behavior," *J. of Complex Systems*, vol. 2, pp. 273–347, 1987.

[98] OSHER, S. and SETHIAN, J. A., "Fronts propagating with curvature-dependent speed: Algorithms based on hamilton-jacobi formulations," *J. of Computational Physics*, vol. 79, pp. 12–49, 1988.

[99] OSTROVSKY, R., RABANI, Y., SCHULMAN, L., and SWAMY, C., "The effectiveness of Lloyd-type methods for the k-means problem," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pp. 165–176, IEEE, 2006.

[100] OZAKIN, A. and GRAY, A., "Submanifold density estimation," in *In Advances in Neural Information Processing Systems (NIPS)*, pp. 1375–1382, 2009.

[101] PAN, J., LAUTERBACH, C., and MANOCHA, D., "Efficient nearest-neighbor computation for GPU-based motion planning," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 2243–2248, IEEE, 2010.

[102] PATIL, A. N., "Distributed multi-probe lsh: Tackling real world data."

[103] PELLETIER, B., "Kernel density estimation on riemannian manifolds," *Statistics and Probability Letters*, vol. 73, pp. 297–304, 2005.

[104] PETER, H. and MARRON, J. S., "Estimation of integrated squared density derivatives," *Statistics and Probability Letters*, vol. 6, pp. 109–115, 1987.

[105] PHAM, D. L., "Fuzzy clustering with spactial contraints," in *Internatinoal Conference on Image Processing*, pp. 65–68, 2002.

[106] POHL, K. M., FISHER, J., GRIMSON, W. E., KIKINIS, R., and WELLS, W. M., "A bayesian model for joint segmentation and registration.," *Neuroimage*, vol. 31, no. 1, p. 228239, 2006.

[107] R., K. and LEE, J., "Navigating nets: Simple algorithms for proximity search," *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA)*, pp. 791–801, 2004.

[108] RAHIMI, A. and RECHT, B., "Random features for large-scale kernel machines.," in *NIPS*, vol. 3, p. 5, 2007.

[109] RAHIMIAN, A., LASHUK, I., VEERAPANENI, S., CHANDRAMOWLISHWARAN, A., MALHOTRA, D., MOON, L., SAMPATH, R., SHRINGARPURE, A., VETTER, J., VUDUC, R., ZORIN, D., and BIROS, G., "Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures," in *SC '10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2010.

[110] RAM, P., LEE, D., MARCH, W., and GRAY, A., "Linear-time algorithms for pairwise statistical problems," *Advances in Neural Information Processing Systems*, vol. 23, 2009.

[111] RIKLIN-RAVIV, T., VAN LEEMPUT, K., MENZE, B. H., WELLS III, W. M., and GOLLAND, P., "Segmentation of image ensembles via latent atlases," *Medical image analysis*, vol. 14, no. 5, p. 654, 2010.

[112] ROUSSON, M. and CREMERS, D., "Efficient kernel density estimation of shape and intensity priors for level set segmentation," in *Medical Image Computing and Computer Assisted Intervention, MICCAI*, pp. 757–764, 2005.

[113] ROWEIS, S. and SAUL, L., "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[114] SAMET, H., *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[115] SANDBERG, B., CHAN, T., and VESE, L., "A level-set and gabor-based active contour algorithm for segmenting textured images," tech. rep., UCLA Department of Mathematics CAM report, 2002.

[116] SCOTT, D. W. and GEORGE, R. T., "Biased and unbiased cross-validation in density estimation," *Journal of the American Statistical Association*, vol. 82, pp. 1131–1146, 1987.

[117] SEZGIN, M. and SANKUR, B., "Survey over image thresholding techniques and quantitative performance evaluation," *J. of Electronic Imaging*, vol. 13, no. 1, pp. 146–168, 2004.

[118] SHAN, S., YANG, P., CHEN, X., and GAO, W., "Adaboost gabor fisher classifier for face recognition," in *Proc. IEEE Int. Workshop Analysis and Modeling of Faces and Gestures*, pp. 278–291, 2005.

[119] SHI, Y., QI, F., XUE, Z., CHEN, L., ITO, K., MATSUO, H., and SHEN, D., "Segmenting lung fields in serial chest radiographs using both population-based and patient-specific shape statistics," *IEEE Transaction on Medical Imaging*, pp. 481–494, 2008.

[120] SILVERMAN, B. W., "Density estimation for statistics and data analysis," 1992.

[121] SILVERMAN, B. W., *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986.

[122] TENENBAUM, J., SILVA, V., and LANGFORD, J., "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[123] TORRALBA, A., FERGUS, R., and FREEMAN, W., "80 million tiny images: A large data set for nonparametric object and scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1958–1970, 2008.

[124] TSAI, A., YEZZI, A., WELLS, W., TEMPANY, C., TUCKER, D., FAN, A., GRIMSON, W., and WILLSKY, A., "A shape-based approach to the segmentation of medical imagery using level sets.," *IEEE Trans. Med. Imaging*, vol. 22, no. 2, p. 137154, 2003.

[125] UHLMANN, J. K., "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letter*, vol. 40, pp. 175–179, 1991.

[126] VOGEL, C., *Computational methods for inverse problems*, vol. 23. SIAM, 2002.

[127] WADHWA, S. and GUPTA, P., "Distributed locality sensitivity hashing," in *Proceedings of the 7th IEEE conference on Consumer communications and networking conference*, CCNC'10, (Piscataway, NJ, USA), pp. 1040–1043, IEEE Press, 2010.

[128] WANG, S., ZHU, W., and LIANG, Z., "Shape deformation: Svm regression and appilcation to medical image segmentation," *International Conf. on Computer Vision*, vol. 2, pp. 209–216, 2001.

[129] WEBER, R., SCHEK, H., and BLOTT, S., "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 194–205, INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS, 1998.

[130] WOLFGANG, H., MARRON, J. S., and WAND, M. P., "Bandwidth choice for density derivatives," *Journal of the Royal Statistical Society*, pp. 223–232, 1990.

[131] WU, B. and NEVATIA, R., "Detection and tracking of multiple, partially occluded humans by bayesian combination of edgelet based part detectors," *IJCV*, pp. 247–266, 2007.

[132] XIAOHUA, C., BRADY, M., LO, J. L.-C., and MORE, N., "Simultaneous segmentation and registration of contrast enhanced breast mri.," *Information Processing in Medical Imaging*, p. 126137, 2005.

[133] XUE, Z., SHEN, D., and DAVATZIKOS, C., "Correspondence detection using wavelet-based attribute vectors," in *Medical Image Computing and Computer Assisted Intervention, MICCAI*, pp. 762–770, 2003.

[134] YANG, C., DURAISWAMI, R., GUMEROV, N. A., and DAVIS, L., "Improved fast gauss transform and efficient kernel density estimation," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pp. 664–671, IEEE, 2003.

[135] YING, L., BIROS, G., and ZORIN, D., "A kernel-independent adaptive fast multipole method in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.

[136] ZHANG, S., ZHAN, Y., DEWAN, M., HUANG, J., METAXAS, D., and ZHOU, X., "Sparse shape composition: a new framework for shape prior modeling," *CVPR*, pp. 1025–1032, 2011.

[137] ZHANG, Y., BRADY, M., and SMITH, S., "Segmentation of brain mr images through a hidden markov field model and the expectation-maximization algorithm," *IEEE Transaction on Medical Imaging*, vol. 20, no. 1, pp. 45–57, 2001.

[138] ZHU, Y., PAPADEMETRIS, X., SINUSAS, A. J., and DUNCAN, J. S., "A dynamical shape prior for lv segmentation from rt3d echocardiography," *MICCAI*, pp. 206–213, 2009.